Journal of Machine Learning Research 22 (2021) 1-64 Submitted 12/19; Published 3/21

Normalizing Flows for Probabilistic Modeling and Inference

George Papamakarios* GPAPAMAK@GOOGLE.COM
Eric Nalisnick* ENALISNICK@GOOGLE.COM
Danilo Jimenez Rezende DANILOR@GOOGLE.COM
Shakir Mohamed SHAKIR@GOOGLE.COM
Balaji Lakshminarayanan BALAJILN@GOOGLE.COM
DeepMind

Editor: Ryan P. Adams

Abstract

Normalizing flows provide a general mechanism for defining expressive probability distributions, only requiring the specification of a (usually simple) base distribution and a series of bijective transformations. There has been much recent work on normalizing flows, ranging from improving their expressive power to expanding their application. We believe the field has now matured and is in need of a unified perspective. In this review, we attempt to provide such a perspective by describing flows through the lens of probabilistic modeling and inference. We place special emphasis on the fundamental principles of flow design, and discuss foundational topics such as expressive power and computational trade-offs. We also broaden the conceptual framing of flows by relating them to more general probability transformations. Lastly, we summarize the use of flows for tasks such as generative modeling, approximate inference, and supervised learning.

Keywords: normalizing flows, invertible neural networks, probabilistic modeling, probabilistic inference, generative models

1. Introduction

The search for well-specified probabilistic models—models that correctly describe the processes that produce data—is one of the enduring ideals of the statistical sciences. Yet, in only the simplest of settings are we able to achieve this goal. A central need in all of statistics and machine learning is then to develop the tools and theories that allow ever-richer probabilistic descriptions to be made, and consequently, that make it possible to develop better-specified models.

This paper reviews one tool we have to address this need: building probability distributions as normalizing flows. Normalizing flows operate by pushing a simple density through a series of transformations to produce a richer, potentially more multi-modal distribution—like a fluid flowing through a set of tubes. As we will see, repeated application of even simple transformations to a unimodal initial density leads to models of exquisite complexity. This

*. Authors contributed equally.

flexibility means that flows are ripe for use in the key statistical tasks of modeling, inference, and simulation.

Normalizing flows are an increasingly active area of machine learning research. Yet there is an absence of a unifying lens with which to understand the latest advancements and their relationships to previous work. The thesis of Papamakarios (2019) and the survey by Kobyzev et al. (2020) have made steps in establishing this broader understanding. Our review complements these existing papers. In particular, our treatment of flows is more comprehensive than Papamakarios (2019)'s but shares some organizing principles. Kobyzev et al. (2020)'s article is commendable in its coverage and synthesis of the literature, discussing both finite and infinitesimal flows (as we do) and curating the latest results in density estimation. Our review is more tutorial in nature and provides in-depth discussion of several areas that Kobyzev et al. (2020) label as open problems (such as extensions to discrete variables and Riemannian manifolds).

Our exploration of normalizing flows attempts to illuminate enduring principles that will guide their construction and application for the foreseeable future. Specifically, our review begins by establishing the formal and conceptual structure of normalizing flows in Section 2. Flow construction is then discussed in detail, both for finite (Section 3) and infinitesimal (Section 4) variants. A more general perspective is then presented in Section 5, which in turn allows for extensions to structured domains and geometries. Lastly, we discuss commonly encountered applications in Section 6.

Notation We use bold symbols to indicate vectors (lowercase) and matrices (uppercase), otherwise variables are scalars. We indicate probabilities by $\Pr(\cdot)$ and probability densities by $p(\cdot)$. We will also use $p(\cdot)$ to refer to the distribution with that density function. We often add a subscript to probability densities—e.g. $p_{\mathrm{x}}(\mathbf{x})$—to emphasize which random variable they refer to. The notation $p(\mathbf{x};\boldsymbol{\theta})$ represents the distribution of random variables $\mathbf{x}$ with distributional parameters $\boldsymbol{\theta}$. The symbol $\nabla_{\boldsymbol{\theta}}$ represents the gradient operator that collects all partial derivatives of a function with respect to parameters in the set $\boldsymbol{\theta}$, that is $\nabla_{\boldsymbol{\theta}} f = [\frac{\partial f}{\partial \theta_1}, \ldots, \frac{\partial f}{\partial \theta_K}]$ for K-dimensional parameters. The Jacobian of a function $f : \mathbb{R}^D \to \mathbb{R}^D$ is denoted by $J_f(\cdot)$. Finally, we represent the sampling or simulation of variates $\mathbf{x}$ from a distribution $p(\mathbf{x})$ using the notation $\mathbf{x} \sim p(\mathbf{x})$.

2. Normalizing Flows

We begin by outlining basic definitions and properties of normalizing flows. We establish the expressive power of flow-based models, explain how to use flows in practice, and provide some historical background. This section doesn't assume prior familiarity with normalizing flows, and can serve as an introduction to the field.

2.1 Definition and Basics

Normalizing flows provide a general way of constructing flexible probability distributions over continuous random variables. Let $\mathbf{x}$ be a D-dimensional real vector, and suppose we

would like to define a joint distribution over $\mathbf{x}$. The main idea of flow-based modeling is to express $\mathbf{x}$ as a transformation T of a real vector $\mathbf{u}$ sampled from $p_{\mathrm{u}}(\mathbf{u})$:

$$\mathbf{x} = T(\mathbf{u}) \quad \text{where} \quad \mathbf{u} \sim p_{\mathrm{u}}(\mathbf{u}). \tag{1}$$

We refer to $p_{\mathrm{u}}(\mathbf{u})$ as the *base distribution* of the flow-based model.[1] The transformation T and the base distribution $p_{\mathrm{u}}(\mathbf{u})$ can have parameters of their own (denote them as $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ respectively); this induces a family of distributions over $\mathbf{x}$ parameterized by $\{\boldsymbol{\phi}, \boldsymbol{\psi}\}$.

The defining property of flow-based models is that the transformation T must be *invertible* and both T and T^{-1} must be *differentiable*. Such transformations are known as *diffeomorphisms* and require that $\mathbf{u}$ be D-dimensional as well (Milnor and Weaver, 1997). Under these conditions, the density of $\mathbf{x}$ is well-defined and can be obtained by a change of variables (Rudin, 2006; Bogachev, 2007):

$$p_{\mathrm{x}}(\mathbf{x}) = p_{\mathrm{u}}(\mathbf{u}) \, |\det J_T(\mathbf{u})|^{-1} \quad \text{where} \quad \mathbf{u} = T^{-1}(\mathbf{x}). \tag{2}$$

Equivalently, we can also write $p_{\mathrm{x}}(\mathbf{x})$ in terms of the Jacobian of T^{-1}:

$$p_{\mathrm{x}}(\mathbf{x}) = p_{\mathrm{u}}\big(T^{-1}(\mathbf{x})\big) \, \left|\det J_{T^{-1}}(\mathbf{x})\right|. \tag{3}$$

The Jacobian $J_T(\mathbf{u})$ is the $D \times D$ matrix of all partial derivatives of T given by:

$$J_T(\mathbf{u}) = \begin{bmatrix} \frac{\partial T_1}{\partial \mathrm{u}_1} & \cdots & \frac{\partial T_1}{\partial \mathrm{u}_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial T_D}{\partial \mathrm{u}_1} & \cdots & \frac{\partial T_D}{\partial \mathrm{u}_D} \end{bmatrix}. \tag{4}$$

In practice, we often construct a flow-based model by implementing T (or T^{-1}) with a neural network and taking $p_{\mathrm{u}}(\mathbf{u})$ to be a simple density such as a multivariate normal. In Sections 3 and 4 we will discuss in detail how to implement T (or T^{-1}).

Intuitively, we can think of the transformation T as warping the space $\mathbb{R}^D$ in order to mold the density $p_{\mathrm{u}}(\mathbf{u})$ into $p_{\mathrm{x}}(\mathbf{x})$. The absolute Jacobian determinant $|\det J_T(\mathbf{u})|$ quantifies the relative change of volume of a small neighbourhood around $\mathbf{u}$ due to T. Roughly speaking, take $\mathbf{du}$ to be an (infinitesimally) small neighbourhood around $\mathbf{u}$ and $\mathbf{dx}$ to be the small neighbourhood around $\mathbf{x}$ that $\mathbf{du}$ maps to. We then have that $|\det J_T(\mathbf{u})| \approx \mathrm{Vol}(\mathbf{dx})/\mathrm{Vol}(\mathbf{du})$, the volume of $\mathbf{dx}$ divided by the volume of $\mathbf{du}$. The probability mass in $\mathbf{dx}$ must equal the probability mass in $\mathbf{du}$. So, if $\mathbf{du}$ is expanded, then the density at $\mathbf{x}$ is smaller than the density at $\mathbf{u}$. If $\mathbf{du}$ is contracted, then the density at $\mathbf{x}$ is larger.

An important property of invertible and differentiable transformations is that they are *composable*. Given two such transformations T_1 and T_2, their composition $T_2 \circ T_1$ is also invertible and differentiable. Its inverse and Jacobian determinant are given by:

$$(T_2 \circ T_1)^{-1} = T_1^{-1} \circ T_2^{-1} \tag{5}$$

$$\det J_{T_2 \circ T_1}(\mathbf{u}) = \det J_{T_2}(T_1(\mathbf{u})) \cdot \det J_{T_1}(\mathbf{u}). \tag{6}$$

1. Some papers refer to $p_{\mathrm{u}}(\mathbf{u})$ as the 'prior' and to $\mathbf{u}$ as the 'latent variable'. We believe that this terminology is not as well-suited for normalizing flows as it is for latent-variable models. Upon observing $\mathbf{x}$, the corresponding $\mathbf{u} = T^{-1}(\mathbf{x})$ is uniquely determined and thus no longer 'latent'.

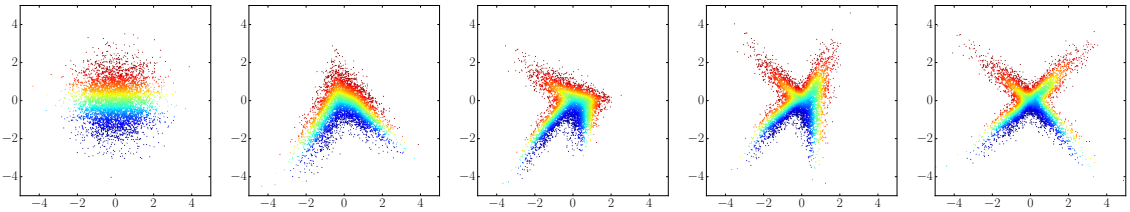


Figure 1: Example of a 4-step flow transforming samples from a standard-normal base density to a cross-shaped target density.

In consequence, we can build complex transformations by composing multiple instances of simpler transformations, without compromising the requirements of invertibility and differentiability, and hence without losing the ability to calculate the density $p_{\mathrm{x}}(\mathbf{x})$.

In practice, it is common to chain together multiple transformations $T_1, \ldots, T_K$ to obtain $T = T_K \circ \cdots \circ T_1$, where each T_k transforms $\mathbf{z}_{k-1}$ into $\mathbf{z}_k$, assuming $\mathbf{z}_0 = \mathbf{u}$ and $\mathbf{z}_K = \mathbf{x}$. Hence, the term 'flow' refers to the trajectory that a collection of samples from $p_{\mathrm{u}}(\mathbf{u})$ follow as they are gradually transformed by the sequence of transformations $T_1, \ldots, T_K$. The term 'normalizing' refers to the fact that the inverse flow through $T_K^{-1}, \ldots, T_1^{-1}$ takes a collection of samples from $p_{\mathrm{x}}(\mathbf{x})$ and transforms them (in a sense, 'normalizes' them) into a collection of samples from a prescribed density $p_{\mathrm{u}}(\mathbf{u})$ (which is often taken to be a multivariate normal). Figure 1 illustrates a flow ($K = 4$) transforming a standard-normal base distribution to a cross-shaped target density.

In terms of functionality, a flow-based model provides two operations: sampling from the model via Equation 1, and evaluating the model's density via Equation 3. These operations have different computational requirements. Sampling from the model requires the ability to sample from $p_{\mathrm{u}}(\mathbf{u})$ and to compute the forward transformation T. Evaluating the model's density requires computing the inverse transformation T^{-1} and its Jacobian determinant, and evaluating the density $p_{\mathrm{u}}(\mathbf{u})$. The application will dictate which of these operations need to be implemented and how efficient they need to be. We discuss the computational trade-offs associated with various implementation choices in Sections 3 and 4.

2.2 Expressive Power of Flow-Based Models

Before discussing the particulars of flows, a question of foremost importance is: how expressive are flow-based models? Can they represent *any* distribution $p_{\mathrm{x}}(\mathbf{x})$, even if the base distribution is restricted to be simple? We show that this universal representation is possible under reasonable conditions on $p_{\mathrm{x}}(\mathbf{x})$. Specifically, we will show that for any pair of well-behaved distributions $p_{\mathrm{x}}(\mathbf{x})$ (the target) and $p_{\mathrm{u}}(\mathbf{u})$ (the base), there exists a diffeomorphism that can turn $p_{\mathrm{u}}(\mathbf{u})$ into $p_{\mathrm{x}}(\mathbf{x})$. The argument is constructive, and is based on a similar proof of the existence of non-linear ICA by Hyvärinen and Pajunen (1999); a more formal treatment is provided by e.g. Bogachev et al. (2005).

Suppose that $p_{\mathrm{x}}(\mathbf{x}) > 0$ for all $\mathbf{x} \in \mathbb{R}^D$, and assume that all conditional probabilities $\Pr(\mathrm{x}'_i \leq \mathrm{x}_i \,|\, \mathbf{x}_{<i})$—with x'_i being the random variable this probability refers to—are differentiable with respect to $(\mathrm{x}_i, \mathbf{x}_{<i})$. Using the chain rule of probability, we can decompose $p_{\mathrm{x}}(\mathbf{x})$ into a product of conditional densities as follows:

$$p_{\mathrm{x}}(\mathbf{x}) = \prod_{i=1}^{D} p_{\mathrm{x}}(\mathrm{x}_i \,|\, \mathbf{x}_{<i}). \tag{7}$$

Since $p_{\mathrm{x}}(\mathbf{x})$ is non-zero everywhere, $p_{\mathrm{x}}(\mathrm{x}_i \,|\, \mathbf{x}_{<i}) > 0$ for all i and $\mathbf{x}$. Next, define the transformation $F : \mathbf{x} \mapsto \mathbf{z} \in (0,1)^D$ whose i-th element is given by the *cumulative distribution function* of the i-th conditional:

$$\mathrm{z}_i = F_i(\mathrm{x}_i, \mathbf{x}_{<i}) = \int_{-\infty}^{\mathrm{x}_i} p_{\mathrm{x}}(\mathrm{x}'_i \,|\, \mathbf{x}_{<i})\, d\mathrm{x}'_i = \Pr(\mathrm{x}'_i \leq \mathrm{x}_i \,|\, \mathbf{x}_{<i}). \tag{8}$$

Since each F_i is differentiable with respect to its inputs, F is differentiable with respect to $\mathbf{x}$. Moreover, each $F_i(\cdot, \mathbf{x}_{<i}) : \mathbb{R} \to (0,1)$ is invertible, since its derivative $\frac{\partial F_i}{\partial \mathrm{x}_i} = p_{\mathrm{x}}(\mathrm{x}_i \,|\, \mathbf{x}_{<i})$ is positive everywhere. Because z_i doesn't depend on x_j for $i < j$, that implies we can invert F, with its inverse F^{-1} given element-by-element as follows:

$$\mathrm{x}_i = (F_i(\cdot, \mathbf{x}_{<i}))^{-1}(\mathrm{z}_i) \quad \text{for } i = 1, \ldots, D. \tag{9}$$

The Jacobian of F is lower triangular since $\frac{\partial F_i}{\partial \mathrm{x}_j} = 0$ for $i < j$. Hence, the Jacobian determinant of F is equal to the product of its diagonal elements:

$$\det J_F(\mathbf{x}) = \prod_{i=1}^{D} \frac{\partial F_i}{\partial \mathrm{x}_i} = \prod_{i=1}^{D} p_{\mathrm{x}}(\mathrm{x}_i \,|\, \mathbf{x}_{<i}) = p_{\mathrm{x}}(\mathbf{x}). \tag{10}$$

Since $p_{\mathrm{x}}(\mathbf{x}) > 0$, the Jacobian determinant is non-zero everywhere. Therefore, the inverse of $J_F(\mathbf{x})$ exists, and is equal to the Jacobian of F^{-1}, so F is a diffeomorphism. Using a change of variables, we can calculate the density of $\mathbf{z}$ as follows:

$$p_{\mathrm{z}}(\mathbf{z}) = p_{\mathrm{x}}(\mathbf{x})\ |\det J_F(\mathbf{x})|^{-1} = p_{\mathrm{x}}(\mathbf{x})\ |p_{\mathrm{x}}(\mathbf{x})|^{-1} = 1, \tag{11}$$

which implies $\mathbf{z}$ is distributed uniformly in the open unit cube $(0,1)^D$.

The above argument shows that a flow-based model can express any distribution $p_{\mathrm{x}}(\mathbf{x})$ (satisfying the conditions stated above) even if we restrict the base distribution to be uniform in $(0,1)^D$. We can extend this statement to any base distribution $p_{\mathrm{u}}(\mathbf{u})$ (satisfying the same conditions as $p_{\mathrm{x}}(\mathbf{x})$) by first transforming $\mathbf{u}$ to a uniform $\mathbf{z} \in (0,1)^D$ as an intermediate step. In particular, given any $p_{\mathrm{u}}(\mathbf{u})$ that satisfies the above conditions, define G to be the following transformation:

$$\mathrm{z}_i = G_i(\mathrm{u}_i, \mathbf{u}_{<i}) = \int_{-\infty}^{\mathrm{u}_i} p_{\mathrm{u}}(\mathrm{u}'_i \,|\, \mathbf{u}_{<i})\, d\mathrm{u}'_i = \Pr(\mathrm{u}'_i \leq \mathrm{u}_i \,|\, \mathbf{u}_{<i}). \tag{12}$$

By the same argument as above, G is a diffeomorphism, and $\mathbf{z}$ is uniformly distributed in $(0,1)^D$. Thus, a flow with transformation $T = F^{-1} \circ G$ can turn $p_{\mathrm{u}}(\mathbf{u})$ into $p_{\mathrm{x}}(\mathbf{x})$.

2.3 Using Flows for Modeling and Inference

Similarly to fitting any probabilistic model, fitting a flow-based model $p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})$ to a target distribution $p^*_{\mathrm{x}}(\mathbf{x})$ can be done by minimizing some divergence or discrepancy between them. This minimization is performed with respect to the model's parameters $\boldsymbol{\theta} = \{\boldsymbol{\phi}, \boldsymbol{\psi}\}$, where $\boldsymbol{\phi}$ are the parameters of T and $\boldsymbol{\psi}$ are the parameters of $p_{\mathrm{u}}(\mathbf{u})$. In the following sections, we discuss a number of divergences for fitting flow-based models, with a particular focus on the Kullback–Leibler (KL) divergence as it is one of the most popular choices.

2.3.1 Forward KL Divergence and Maximum Likelihood Estimation

The forward KL divergence between the target distribution $p^*_{\mathrm{x}}(\mathbf{x})$ and the flow-based model $p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})$ can be written as follows:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= D_{\mathrm{KL}}\left[\, p^*_{\mathrm{x}}(\mathbf{x}) \,\|\, p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta}) \,\right] \\ &= -\mathbb{E}_{p^*_{\mathrm{x}}(\mathbf{x})}\left[\, \log p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta}) \,\right] + \text{const.} \\ &= -\mathbb{E}_{p^*_{\mathrm{x}}(\mathbf{x})}\left[\, \log p_{\mathrm{u}}\big(T^{-1}(\mathbf{x};\boldsymbol{\phi});\boldsymbol{\psi}\big) + \log\left|\det J_{T^{-1}}(\mathbf{x};\boldsymbol{\phi})\right| \,\right] + \text{const.}\end{aligned} \tag{13}$$

The forward KL divergence is well-suited for situations in which we have samples from the target distribution (or the ability to generate them), but we cannot necessarily evaluate the target density $p^*_{\mathrm{x}}(\mathbf{x})$. Assuming we have a set of samples $\{\mathbf{x}_n\}_{n=1}^N$ from $p^*_{\mathrm{x}}(\mathbf{x})$, we can estimate the expectation over $p^*_{\mathrm{x}}(\mathbf{x})$ by Monte Carlo as follows:

$$\mathcal{L}(\boldsymbol{\theta}) \approx -\frac{1}{N}\sum_{n=1}^{N} \log p_{\mathrm{u}}(T^{-1}(\mathbf{x}_n;\boldsymbol{\phi});\boldsymbol{\psi}) + \log\left|\det J_{T^{-1}}(\mathbf{x}_n;\boldsymbol{\phi})\right| + \text{const.} \tag{14}$$

Minimizing the above Monte Carlo approximation of the KL divergence is equivalent to fitting the flow-based model to the samples $\{\mathbf{x}_n\}_{n=1}^N$ by maximum likelihood estimation.

In practice, we often optimize the parameters $\boldsymbol{\theta}$ iteratively with stochastic gradient-based methods. We can obtain an unbiased estimate of the gradient of the KL divergence with respect to the parameters as follows:

$$\nabla_{\boldsymbol{\phi}}\mathcal{L}(\boldsymbol{\theta}) \approx -\frac{1}{N}\sum_{n=1}^{N} \nabla_{\boldsymbol{\phi}} \log p_{\mathrm{u}}\big(T^{-1}(\mathbf{x}_n;\boldsymbol{\phi});\boldsymbol{\psi}\big) + \nabla_{\boldsymbol{\phi}} \log\left|\det J_{T^{-1}}(\mathbf{x}_n;\boldsymbol{\phi})\right| \tag{15}$$

$$\nabla_{\boldsymbol{\psi}}\mathcal{L}(\boldsymbol{\theta}) \approx -\frac{1}{N}\sum_{n=1}^{N} \nabla_{\boldsymbol{\psi}} \log p_{\mathrm{u}}\big(T^{-1}(\mathbf{x}_n;\boldsymbol{\phi});\boldsymbol{\psi}\big). \tag{16}$$

The update with respect to $\boldsymbol{\psi}$ may also be done in closed form if $p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$ admits closed-form maximum likelihood estimates, as is the case for example with Gaussian distributions.

In order to fit a flow-based model via maximum likelihood, we need to compute T^{-1}, its Jacobian determinant and the density $p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$—as well as differentiate through all three, if using gradient-based optimization. That means we can train a flow model with maximum likelihood even if we are not able to compute T or sample from $p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$. Yet these operations will be needed if we want to sample from the model after it is fitted.

2.3.2 Reverse KL Divergence

Alternatively, we may fit the flow-based model by minimizing the reverse KL divergence, which can be written as follows:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= D_{\mathrm{KL}}\left[\, p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta}) \,\|\, p^*_{\mathrm{x}}(\mathbf{x}) \,\right] \\ &= \mathbb{E}_{p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})}\left[\, \log p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta}) - \log p^*_{\mathrm{x}}(\mathbf{x}) \,\right] \\ &= \mathbb{E}_{p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})}\left[\, \log p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi}) - \log \left|\det J_T(\mathbf{u};\boldsymbol{\phi})\right| - \log p^*_{\mathrm{x}}(T(\mathbf{u};\boldsymbol{\phi})) \,\right].\end{aligned} \tag{17}$$

We made use of a change of variable in order to express the expectation with respect to $\mathbf{u}$. The reverse KL divergence is suitable when we have the ability to evaluate the target density $p^*_{\mathrm{x}}(\mathbf{x})$ but not necessarily sample from it. In fact, we can minimize $\mathcal{L}(\boldsymbol{\theta})$ even if we can only evaluate $p^*_{\mathrm{x}}(\mathbf{x})$ up to a multiplicative normalizing constant C, since in that case $\log C$ will be an additive constant in the above expression for $\mathcal{L}(\boldsymbol{\theta})$. We may therefore assume that $p^*_{\mathrm{x}}(\mathbf{x}) = \widetilde{p}_{\mathrm{x}}(\mathbf{x})/C$, where $\widetilde{p}_{\mathrm{x}}(\mathbf{x})$ is tractable but $C = \int \widetilde{p}_{\mathrm{x}}(\mathbf{x})\, d\mathbf{x}$ is not, and rewrite the reverse KL divergence as:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})}\left[\, \log p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi}) - \log \left|\det J_T(\mathbf{u};\boldsymbol{\phi})\right| - \log \widetilde{p}_{\mathrm{x}}(T(\mathbf{u};\boldsymbol{\phi})) \,\right] + \text{const.} \tag{18}$$

In practice, we can minimize $\mathcal{L}(\boldsymbol{\theta})$ iteratively with stochastic gradient-based methods. Since we reparameterized the expectation to be with respect to the base distribution $p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$, we can easily obtain an unbiased estimate of the gradient of $\mathcal{L}(\boldsymbol{\theta})$ with respect to $\boldsymbol{\phi}$ by Monte Carlo. In particular, let $\{\mathbf{u}_n\}_{n=1}^N$ be a set of samples from $p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$; the gradient of $\mathcal{L}(\boldsymbol{\theta})$ with respect to $\boldsymbol{\phi}$ can be estimated as follows:

$$\nabla_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}) \approx -\frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\phi}} \log \left|\det J_T(\mathbf{u}_n;\boldsymbol{\phi})\right| + \nabla_{\boldsymbol{\phi}} \log \widetilde{p}_{\mathrm{x}}(T(\mathbf{u}_n;\boldsymbol{\phi})). \tag{19}$$

Similarly, we can estimate the gradient with respect to $\boldsymbol{\psi}$ by reparameterizing $\mathbf{u}$ as:

$$\mathbf{u} = T'(\mathbf{u}';\boldsymbol{\psi}) \quad \text{where} \quad \mathbf{u}' \sim p_{\mathrm{u}'}(\mathbf{u}'), \tag{20}$$

and then writing the expectation with respect to $p_{\mathrm{u}'}(\mathbf{u}')$. However, since we can equivalently absorb the reparameterization T' into T and replace the base distribution with $p_{\mathrm{u}'}(\mathbf{u}')$, we can assume without loss of generality that the parameters $\boldsymbol{\psi}$ are fixed and only optimize with respect to $\boldsymbol{\phi}$.

In order to minimize the reverse KL divergence as described above, we need to be able to sample from the base distribution $p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$ as well as compute and differentiate through the transformation T and its Jacobian determinant. That means that we can fit a flow-based model by minimizing the reverse KL divergence even if we cannot evaluate the base density or compute the inverse transformation T^{-1}. However, we will need these operations if we would like to evaluate the density of the trained model.

The reverse KL divergence is often used for *variational inference* (Wainwright and Jordan, 2008; Blei et al., 2017), a form of approximate Bayesian inference. In this case, the target is the posterior, making $\widetilde{p}_{\mathrm{x}}(\mathbf{x})$ the product between a likelihood function and a prior density.

Examples of work using flows in variational inference are given by Rezende and Mohamed (2015); van den Berg et al. (2018); Kingma et al. (2016); Tomczak and Welling (2016); Louizos and Welling (2017). We cover this topic in more detail in Section 6.2.3.

Another application of the reverse KL divergence is in the context of *model distillation*: a flow model is trained to replace a target model $p^*_{\mathrm{x}}(\mathbf{x})$ whose density can be evaluated but that is otherwise inconvenient. An example of model distillation with flows is given by van den Oord et al. (2018). In their case, samples cannot be efficiently drawn from the target model and so they distill it into a flow that supports fast sampling.

2.3.3 Relationship Between Forward and Reverse KL Divergence

An alternative perspective of a flow-based model is to think of the target $p^*_{\mathrm{x}}(\mathbf{x})$ as the *base distribution* and the inverse flow as inducing a distribution $p^*_{\mathrm{u}}(\mathbf{u};\boldsymbol{\phi})$. Intuitively, $p^*_{\mathrm{u}}(\mathbf{u};\boldsymbol{\phi})$ is the distribution that the training data would follow when passed through T^{-1}. Since the target and the base distributions uniquely determine each other given the transformation between them, $p^*_{\mathrm{u}}(\mathbf{u};\boldsymbol{\phi}) = p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$ if and only if $p^*_{\mathrm{x}}(\mathbf{x}) = p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})$. Therefore, fitting the model $p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})$ to the target $p^*_{\mathrm{x}}(\mathbf{x})$ can be equivalently thought of as fitting the induced distribution $p^*_{\mathrm{u}}(\mathbf{u};\boldsymbol{\phi})$ to the base $p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$.

We may now ask: how does fitting $p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})$ to the target relate to fitting $p^*_{\mathrm{u}}(\mathbf{u};\boldsymbol{\phi})$ to the base? Using a change of variables (see Section A for details), we can show the following equality (Papamakarios et al., 2017):

$$D_{\mathrm{KL}}\left[\,p^*_{\mathrm{x}}(\mathbf{x})\,\|\,p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})\,\right] = D_{\mathrm{KL}}\left[\,p^*_{\mathrm{u}}(\mathbf{u};\boldsymbol{\phi})\,\|\,p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})\,\right]. \tag{21}$$

The above equality means that fitting the model to the target using the forward KL divergence (maximum likelihood) is equivalent to fitting the induced distribution $p^*_{\mathrm{u}}(\mathbf{u};\boldsymbol{\phi})$ to the base $p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})$ under the reverse KL divergence. In Section A, we also show that:

$$D_{\mathrm{KL}}\left[\,p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})\,\|\,p^*_{\mathrm{x}}(\mathbf{x})\,\right] = D_{\mathrm{KL}}\left[\,p_{\mathrm{u}}(\mathbf{u};\boldsymbol{\psi})\,\|\,p^*_{\mathrm{u}}(\mathbf{u};\boldsymbol{\phi})\,\right], \tag{22}$$

which means that fitting the model to the target via the reverse KL divergence is equivalent to fitting $p^*_{\mathrm{u}}(\mathbf{u};\boldsymbol{\phi})$ to the base via the forward KL divergence (maximum likelihood).

2.3.4 Alternative Divergences

Learning the parameters of flow models is not restricted to the use of the KL divergence. Many alternative measures of difference between distributions are available. These alternatives are often grouped into two general families, the *f-divergences* that use density ratios to compare distributions, and the *integral probability metrics* (IPMs) that use differences for comparison:

$$f\text{-divergence} \qquad D_f\left[\,p^*_{\mathrm{x}}(\mathbf{x})\,\|\,p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})\,\right] = \mathbb{E}_{p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})}\left[\,f\left(\frac{p^*_{\mathrm{x}}(\mathbf{x})}{p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})}\right)\right] \tag{23}$$

$$\text{IPM} \qquad \delta_s\left[\,p^*_{\mathrm{x}}(\mathbf{x})\,\|\,p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})\,\right] = \mathbb{E}_{p^*_{\mathrm{x}}(\mathbf{x})}\left[\,s(\mathbf{x})\,\right] - \mathbb{E}_{p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})}\left[\,s(\mathbf{x})\,\right]. \tag{24}$$

For the f-divergences, the function f is convex; when this function is $f(r) = r \log r$ we recover the KL divergence. For IPMs, the function s can be chosen from a set of test statistics, or can be a witness function chosen adversarially.

The same considerations that applied to the KL divergence previously apply here and inform the choice of divergence: can we simulate from the model $p_{\mathrm{x}}(\mathbf{x}; \boldsymbol{\theta})$, do we know the true distribution up to a multiplicative constant, do we have access to the transform or its inverse? When considering these divergences, we see a connection between flow-based models, whose design principles use composition and change of variables, to the more general class of implicit probabilistic models (Diggle and Gratton, 1984; Mohamed and Lakshminarayanan, 2016). If we choose the generator of a generative adversarial network as a normalizing flow, we can train the flow parameters using adversarial training (Grover et al., 2018; Danihelka et al., 2017), with Wasserstein losses (Arjovsky et al., 2017), using maximum mean discrepancy (Bińkowski et al., 2018), or other approaches.

2.4 Brief Historical Overview

Whitening transformations (Johnson, 1966; Friedman, 1987)—so-called as they transform data into white noise—are the clearest intellectual predecessor to the use of normalizing flows within machine learning. Chen and Gopinath (2000) were perhaps the first to use whitening as a density estimation technique rather than for feature pre-processing, calling the method *Gaussianization*. Tabak and Vanden-Eijnden (2010) approached Gaussianization from the perspective of diffusion processes, making connections to statistical mechanics—specifically, using Liouville's equation to characterize the flow. In a follow-up paper, Tabak and Turner (2013) introduce what can be thought of as the modern conception of normalizing flows: introducing the very term *normalizing flow* and defining the flow generally as a composition of K simple maps. As we will discuss in Section 3, this definition via composition is essential for enabling flows to be expressive while preserving computational and analytical tractability.

The idea of composition saw its recent emergence in machine learning starting with Rippel and Adams (2013), who were perhaps the first to recognize that parameterizing flows with deep neural networks could result in quite general and expressive distribution classes. Dinh et al. (2015) then introduced a scalable and computationally efficient architecture, demonstrating further improvements to image modeling and inference. Rezende and Mohamed (2015) used the idea and language from Tabak and Turner (2013) to apply normalizing flows in the setting of variational inference. Following this, as the papers that are reviewed here will show, normalizing flows now constitute a broad literature in their own right, with much work expanding the applicability and scalability of these initial efforts.

Because of the connection to change of variables, we can also connect the current use of normalizing flows in machine learning with its pre-history in many other settings. The change of measure has been studied intensely in the development of statistical mechanics—a famous example being the aforementioned *Liouville's theorem* (1838). *Copulas* (Sklar, 1959; Elidan, 2013) can be viewed as rudimentary flows, where each dimension is transformed independently using the empirically-estimated marginal cumulative distribution function.

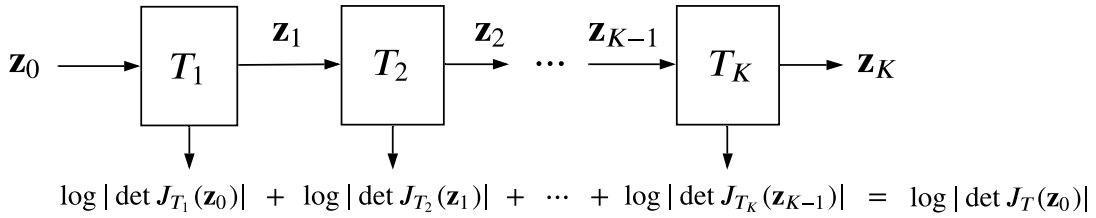


Figure 2: Illustration of a flow composed of K transformations.

Optimal transport and the Wasserstein metric (Villani, 2008) can also be formulated in terms of transformations of measures ('transport of measures')—also known as the *Monge problem*. In particular, *triangular maps* (a concept deeply related to autoregressive flows) can be shown to be a limiting solution to a class of Monge–Kantorovich problems (Carlier et al., 2010). This class of triangular maps itself has a long history, with Rosenblatt (1952) studying their properties for transforming multivariate distributions uniformly over the hypercube. Optimal transport could be a tutorial unto itself and therefore we mostly sidestep this framework, instead choosing to think in terms of the change of variables.

3. Constructing Flows Part I: Finite Compositions

Having described some high-level properties and uses of flows, we transition into describing, categorizing, and unifying the various ways to construct a flow. As discussed in Section 2.1, normalizing flows are *composable*; that is, we can construct a flow with transformation T by composing a finite number of simple transformations T_k as follows:

$$T = T_K \circ \cdots \circ T_1. \tag{25}$$

The idea is to use simple transformations as building blocks—each having a tractable inverse and Jacobian determinant—to define a complex transformation with more expressive power than any of its constituent components. Importantly, the flow's forward and inverse evaluation and Jacobian-determinant computation can be localized to the sub-flows. As illustrated in Figure 2, assuming $\mathbf{z}_0 = \mathbf{u}$ and $\mathbf{z}_K = \mathbf{x}$, the forward evaluation is:

$$\mathbf{z}_k = T_k(\mathbf{z}_{k-1}) \quad \text{for } k = 1, \dots, K, \tag{26}$$

the inverse evaluation is:

$$\mathbf{z}_{k-1} = T_k^{-1}(\mathbf{z}_k) \quad \text{for } k = K, \dots, 1, \tag{27}$$

and the Jacobian-determinant computation (in the log domain) is:

$$\log|\det J_T(\mathbf{z}_0)| = \log\left|\prod_{k=1}^{K} \det J_{T_k}(\mathbf{z}_{k-1})\right| = \sum_{k=1}^{K} \log|\det J_{T_k}(\mathbf{z}_{k-1})|\,. \tag{28}$$

Autoregressive flows	Transformer type: – Affine – Combination-based – Integration-based – Spline-based	Conditioner type: – Recurrent – Masked – Coupling layer
Linear flows	Permutations Decomposition-based: – PLU – QR Orthogonal: – Exponential map – Cayley map – Householder	
Residual flows	Contractive residual Based on matrix determinant lemma: – Planar – Sylvester – Radial	

Table 1: Overview of methods for constructing flows based on finite compositions.

Increasing the 'depth' (i.e. number of composed sub-flows) of the transformation crucially results in only $\mathcal{O}(K)$ growth in the computational complexity—a pleasantly practical cost to pay for the increased expressivity.

In practice we implement either T_k or T_k^{-1} using a model (such as a neural network) with parameters $\boldsymbol{\phi}_k$, which we will denote as $f_{\boldsymbol{\phi}_k}$. That is, we can take the model $f_{\boldsymbol{\phi}_k}$ to implement either T_k, in which case it will take in $\mathbf{z}_{k-1}$ and output $\mathbf{z}_k$, or T_k^{-1}, in which case it will take in $\mathbf{z}_k$ and output $\mathbf{z}_{k-1}$. In either case, we must ensure that the model is invertible and has a tractable Jacobian determinant. In the rest of this section, we will describe several approaches for constructing $f_{\boldsymbol{\phi}_k}$ so that these requirements are satisfied. An overview of all the methods discussed in this section is shown in Table 1.

Ensuring that $f_{\boldsymbol{\phi}_k}$ is invertible and explicitly calculating its inverse are *not* synonymous. In many implementations, even though the inverse of $f_{\boldsymbol{\phi}_k}$ is guaranteed to exist, it can be expensive or even intractable to compute exactly. As discussed in Section 2, the forward transformation T is used when sampling, and the inverse transformation T^{-1} is used when evaluating densities. If the inverse of $f_{\boldsymbol{\phi}_k}$ is not efficient, either density evaluation or sampling will be inefficient or even intractable, depending on whether $f_{\boldsymbol{\phi}_k}$ implements T_k or T_k^{-1}. Whether $f_{\boldsymbol{\phi}_k}$ should be designed to have an efficient inverse and whether it should be taken to implement T_k or T_k^{-1} are decisions that ought to be based on intended usage.

We should also clarify what we mean by 'tractable Jacobian determinant'. We can always compute the Jacobian matrix of a differentiable function with D inputs and D outputs, using D passes of either forward-mode or reverse-mode automatic differentiation. Then, we can explicitly calculate the determinant of that Jacobian. However, this computation has a time cost of $\mathcal{O}(D^3)$, which can be intractable for large D. For most applications of flow-based models, the Jacobian-determinant computation should be at most $\mathcal{O}(D)$. Hence, in the following sections, we will describe functional forms that allow the Jacobian determinant to be computed in linear time with respect to the input dimensionality.

To simplify notation from here on, we will drop the dependence of the model parameters on k and denote the model by $f_{\boldsymbol{\phi}}$. Also, we will denote the model's input by $\mathbf{z}$ and its output by $\mathbf{z}'$, regardless of whether the model implements T_k or T_k^{-1}.

3.1 Autoregressive Flows

Autoregressive flows were one of the first classes of flows developed and remain among the most popular. In Section 2.2 we saw that, under mild conditions, we can transform any distribution $p_{\mathrm{x}}(\mathbf{x})$ into a uniform distribution in $(0,1)^D$ using maps with a triangular Jacobian. Autoregressive flows are a direct implementation of this construction, specifying $f_{\boldsymbol{\phi}}$ to have the following form (as described by e.g. Huang et al., 2018; Jaini et al., 2019):

$$\mathrm{z}'_i = \tau(\mathrm{z}_i; \boldsymbol{h}_i) \quad \text{where} \quad \boldsymbol{h}_i = c_i(\mathbf{z}_{<i}), \tag{29}$$

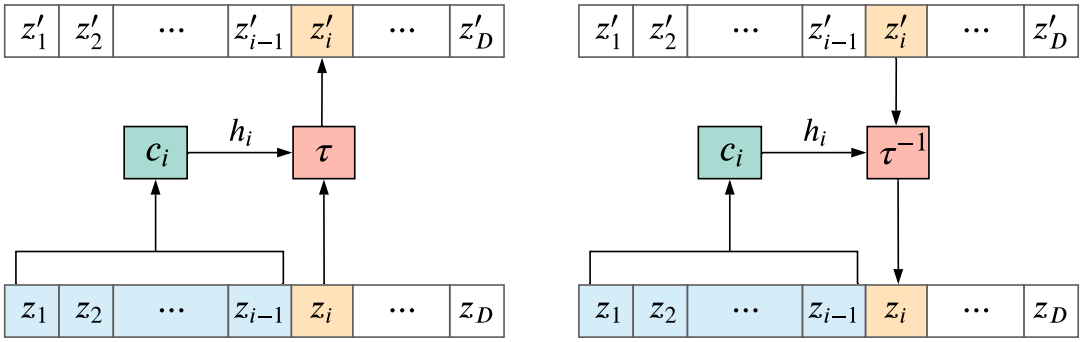
where τ is termed the *transformer* and c_i the i-th *conditioner*. This is illustrated in Figure 3a. The transformer is a strictly monotonic function of z_i (and therefore invertible), is parameterized by $\boldsymbol{h}_i$, and specifies how the flow acts on z_i in order to output z'_i. The conditioner determines the parameters of the transformer, and in turn, can modify the transformer's behavior. The conditioner does *not* need to be a bijection. Its one constraint is that the i-th conditioner can take as input only the variables with dimension indices less that i. The parameters $\boldsymbol{\phi}$ of $f_{\boldsymbol{\phi}}$ are typically the parameters of the conditioner (not shown above for notational simplicity), but sometimes the transformer has its own parameters too (in addition to $\boldsymbol{h}_i$).

It is easy to check that the above construction is invertible for any choice of τ and c_i as long as the transformer is invertible. Given $\mathbf{z}'$, we can compute $\mathbf{z}$ iteratively as follows:

$$\mathrm{z}_i = \tau^{-1}(\mathrm{z}'_i; \boldsymbol{h}_i) \quad \text{where} \quad \boldsymbol{h}_i = c_i(\mathbf{z}_{<i}). \tag{30}$$

This is illustrated in Figure 3b. In the forward computation, each $\boldsymbol{h}_i$ and therefore each z'_i can be computed independently in any order or in parallel. In the inverse computation however, all $\mathbf{z}_{<i}$ need to have been computed *before* z_i, so that $\mathbf{z}_{<i}$ is available to the conditioner for computing $\boldsymbol{h}_i$.

It is also easy to show that the Jacobian of the above transformation is *triangular*, and thus the Jacobian determinant is tractable. Since each z'_i doesn't depend on $\mathbf{z}_{>i}$, the partial derivative of z'_i with respect to z_j is zero whenever $j > i$. Hence, the Jacobian of $f_{\boldsymbol{\phi}}$ can be



(a) Forward (b) Inverse

Figure 3: Illustration of the i-th step of an autoregressive flow.

written in the following form:

$$J_{f_\phi}(\mathbf{z}) = \begin{bmatrix} \frac{\partial \tau}{\partial z_1}(z_1; \boldsymbol{h}_1) & & \mathbf{0} \\ & \ddots & \\ \mathbf{L}(\mathbf{z}) & & \frac{\partial \tau}{\partial z_D}(z_D; \boldsymbol{h}_D) \end{bmatrix}. \tag{31}$$

The Jacobian is a lower-triangular matrix whose diagonal elements are the derivatives of the transformer for each of the D elements of $\mathbf{z}$. Since the determinant of any triangular matrix is equal to the product of its diagonal elements, the log-absolute-determinant of $J_{f_\phi}(\mathbf{z})$ can be calculated in $\mathcal{O}(D)$ time as follows:

$$\log \left| \det J_{f_\phi}(\mathbf{z}) \right| = \log \left| \prod_{i=1}^{D} \frac{\partial \tau}{\partial z_i}(z_i; \boldsymbol{h}_i) \right| = \sum_{i=1}^{D} \log \left| \frac{\partial \tau}{\partial z_i}(z_i; \boldsymbol{h}_i) \right|. \tag{32}$$

The lower-triangular part of the Jacobian— denoted here by $\mathbf{L}(\mathbf{z})$—is irrelevant. The derivatives of the transformer can be computed either analytically or via automatic differentiation, depending on the implementation.

Autoregressive flows are *universal approximators* (under the conditions discussed in Section 2.2) provided the transformer and the conditioner are flexible enough to represent any function arbitrarily well. This follows directly from the fact that the universal transformation from Section 2.2, which is based on the cumulative distribution functions of the conditionals, is indeed an autoregressive flow. Yet, this is just a statement of representational power and makes no guarantees about the flow's behavior in practice.

An alternative, but mathematically equivalent, formulation of autoregressive flows is to have the conditioner c_i take in $\mathbf{z}'_{<i}$ instead of $\mathbf{z}_{<i}$. This is equivalent to swapping τ with τ^{-1} and $\mathbf{z}$ with $\mathbf{z}'$ in the formulation presented above. Both formulations are common in the literature; here we use the convention that c_i takes in $\mathbf{z}_{<i}$ without loss of generality. The computational differences between the two alternatives are discussed in more detail by e.g. Kingma et al. (2016); Papamakarios et al. (2017).

Implementing an autoregressive flow boils down to (a) implementing the transformer τ and (b) implementing the conditioner c_i. These are independent choices: any type of transformer can be paired up with any type of conditioner, yielding the various combinations that have appeared in the literature. In the following paragraphs, we will list a number of transformer implementations (Section 3.1.1) and an number of conditioner implementations (Section 3.1.2). We will discuss their pros and cons, and point out the choices that have been used by the specific models in the literature.

3.1.1 Implementing the Transformer

Affine transformers One of the simplest possible choices for the transformer—and one of the first used—is the class of affine functions:

$$\tau(\mathrm{z}_i; \boldsymbol{h}_i) = \alpha_i \mathrm{z}_i + \beta_i \quad \text{where} \quad \boldsymbol{h}_i = \{\alpha_i, \beta_i\}\,. \tag{33}$$

The above can be thought of as a *location-scale transformation*, where α_i controls the scale and β_i controls the location. Invertibility is guaranteed if $\alpha_i \neq 0$, and this can be easily achieved by e.g. taking $\alpha_i = \exp \tilde{\alpha}_i$, where $\tilde{\alpha}_i$ is an unconstrained parameter (in which case $\boldsymbol{h}_i = \{\tilde{\alpha}_i, \beta_i\}$). The derivative of the transformer with respect to z_i is equal to α_i; hence the log absolute Jacobian determinant is:

$$\log \left| \det J_{f_\phi}(\mathbf{z}) \right| = \sum_{i=1}^{D} \log |\alpha_i| = \sum_{i=1}^{D} \tilde{\alpha}_i. \tag{34}$$

Autoregressive flows with affine transformers are attractive because of their simplicity and analytical tractability, but their expressivity is limited. To illustrate why, suppose $\mathbf{z}$ follows a Gaussian distribution; then, each z'_i conditioned on $\mathbf{z}'_{<i}$ will also follow a Gaussian distribution. In other words, a single affine autoregressive transformation of a multivariate Gaussian results in a distribution whose conditionals $p_{\mathrm{z}'}(\mathrm{z}'_i \,|\, \mathbf{z}'_{<i})$ will necessarily be Gaussian. Nonetheless, expressive flows can still be obtained by stacking multiple affine autoregressive layers, but it's unknown whether affine autoregressive flows with multiple layers are universal approximators or not. Affine transformers are popular in the literature, having been used in models such as NICE (Dinh et al., 2015), Real NVP (Dinh et al., 2017), IAF (Kingma et al., 2016), MAF (Papamakarios et al., 2017), and Glow (Kingma and Dhariwal, 2018).

Combination-based transformers Non-affine transformers can be constructed from simple components based on the observation that *conic combinations* as well as *compositions* of monotonic functions are also monotonic. Given monotonic functions $\tau_1, \ldots, \tau_K$ of a real variable z, the following functions are also monotonic:

- Conic combination: $\tau(\mathrm{z}) = \sum_{k=1}^{K} w_k \tau_k(\mathrm{z})$, where $w_k > 0$ for all k.
- Composition: $\tau(\mathrm{z}) = \tau_K \circ \cdots \circ \tau_1(\mathrm{z})$.

For example, a non-affine transformer can be constructed using a conic combination of monotonically increasing activation functions $\sigma(\cdot)$ (such as logistic sigmoid, tanh, leaky

ReLU (Maas et al., 2013), and others):

$$\tau(\mathrm{z}_i; \boldsymbol{h}_i) = w_{i0} + \sum_{k=1}^{K} w_{ik}\sigma(\alpha_{ik}\mathrm{z}_i + \beta_{ik}) \quad \text{where} \quad \boldsymbol{h}_i = \{w_{i0}, \ldots, w_{iK}, \alpha_{ik}, \beta_{ik}\}\,, \tag{35}$$

provided $\alpha_{ik} > 0$ and $w_{ik} > 0$ for all $k \geq 1$. Clearly, the above construction corresponds to a monotonic *single-layer perceptron*. By repeatedly combining and composing monotonic activation functions, we can construct a *multi-layer perceptron* that is monotonic, provided that all its weights are strictly positive.

Transformers such as the above can represent any monotonic function arbitrarily well, which follows directly from the universal-approximation capabilities of multi-layer perceptrons (see e.g. Huang et al., 2018, for details). The derivatives of the transformer needed for the computation of the Jacobian determinant are in principle analytically obtainable, but more commonly they are computed via backpropagation. A drawback of combination-based transformers is that in general they cannot be inverted analytically, and can be inverted only iteratively e.g. using bisection search (Burden and Faires, 1989). Variants of combination-based transformers have been used in models such as NAF (Huang et al., 2018), block-NAF (De Cao et al., 2019), and Flow++ (Ho et al., 2019).

Integration-based transformers Another way to define a non-affine transformer is by recognizing that the integral of some positive function is a monotonically increasing function. For example, Wehenkel and Louppe (2019) define the transformer as:

$$\tau(\mathrm{z}_i; \boldsymbol{h}_i) = \int_0^{\mathrm{z}_i} g(\mathrm{z}; \boldsymbol{\alpha}_i)\, d\mathrm{z} + \beta_i \quad \text{where} \quad \boldsymbol{h}_i = \{\boldsymbol{\alpha}_i, \beta_i\}\,, \tag{36}$$

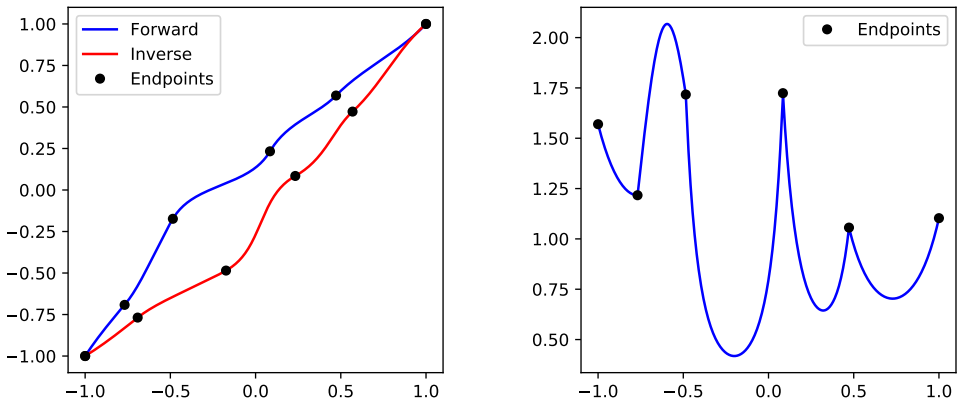
where $g(\cdot; \boldsymbol{\alpha}_i)$ can be any positive-valued neural network parameterized by $\boldsymbol{\alpha}_i$. Typically $g(\cdot; \boldsymbol{\alpha}_i)$ will have its own parameters in addition to $\boldsymbol{\alpha}_i$. The derivative of the transformer required for the computation of the Jacobian determinant is simply equal to $g(\mathrm{z}_i; \boldsymbol{\alpha}_i)$. This approach results in arbitrarily flexible transformers, but the integral lacks analytical tractability. One possibility is to resort to a numerical approximation.

An analytically tractable integration-based transformer can be obtained by taking the integrand $g(\cdot; \boldsymbol{\alpha}_i)$ to be a positive polynomial of degree $2L$. The integral will be a polynomial of degree $2L+1$ in z_i, and thus can be computed analytically. Since every positive polynomial of degree $2L$ can be written as a sum of 2 (or more) squares of polynomials of degree L (Marshall, 2008, Proposition 1.1.2), this fact can be exploited to define a *sum-of-squares polynomial transformer* (Jaini et al., 2019):

$$\tau(\mathrm{z}_i; \boldsymbol{h}_i) = \int_0^{\mathrm{z}_i} \sum_{k=1}^{K} \left(\sum_{\ell=0}^{L} \alpha_{ik\ell}\, \mathrm{z}^\ell \right)^2 d\mathrm{z} + \beta_i, \tag{37}$$

where $\boldsymbol{h}_i$ comprises β_i and all polynomial coefficients $\alpha_{ik\ell}$, and $K \geq 2$. A nice property of the sum-of-squares polynomial transformer is that the coefficients $\alpha_{ik\ell}$ are unconstrained. Moreover, the affine transformer can be derived as the special case of $L = 0$:

$$\int_0^{\mathrm{z}_i} \sum_{k=1}^{K} \left(\alpha_{ik0}\, \mathrm{z}^0\right)^2 d\mathrm{z} + \beta_i = \left(\sum_{k=1}^{K} \alpha_{ik0}^2 \right) \mathrm{z} \Big|_0^{\mathrm{z}_i} + \beta_i = \alpha_i \mathrm{z}_i + \beta_i, \tag{38}$$



(a) Forward and inverse transformer

(b) Transformer derivative

Figure 4: Example of a spline-based transformer with 5 segments. Each segment is a monotonic rational-quadratic function, which can be easily inverted (Durkan et al., 2019b). The locations of the endpoints (black dots) parameterize the spline.

where $\alpha_i = \sum_{k=1}^{K} \alpha_{ik0}^2$. It can be shown that, for large enough L, the sum-of-squares polynomial transformer can approximate arbitrarily well any monotonically increasing function (Jaini et al., 2019, Theorem 3). Nonetheless, since only polynomials of degree up to 4 can be solved analytically, the sum-of-squares polynomial transformer is not analytically invertible for $L \geq 2$, and can only be inverted iteratively using e.g. bisection search (Burden and Faires, 1989).

Spline-based transformers So far, we have discussed non-affine transformers that can be made arbitrarily flexible, but don't have an analytic inverse. One way to overcome this limitation is by implementing the transformer as a monotonic *spline*, i.e. a piecewise function consisting of K segments, where each segment is a simple function that is easy to invert. Specifically, given a set of $K+1$ input locations $\mathrm{z}_{i0}, \ldots, \mathrm{z}_{iK}$, the transformer $\tau(\mathrm{z}_i; \boldsymbol{h}_i)$ is taken to be a simple monotonic function (e.g. a low-degree polynomial) in each interval $[\mathrm{z}_{i(k-1)}, \mathrm{z}_{ik}]$, under the constraint that the K segments meet at the endpoints $\mathrm{z}_{i1}, \ldots, \mathrm{z}_{i(K-1)}$. Outside the interval $[\mathrm{z}_{i0}, \mathrm{z}_{iK}]$, the transformer can default to a simple function such as the identity. Typically, the parameters $\boldsymbol{h}_i$ of the transformer are the input locations $\mathrm{z}_{i0}, \ldots, \mathrm{z}_{iK}$, the corresponding output locations $\mathrm{z}'_{i0}, \ldots, \mathrm{z}'_{iK}$, and (depending on the type of spline) the derivatives (i.e. slopes) at $\mathrm{z}_{i0}, \ldots, \mathrm{z}_{iK}$. See Figure 4 for an illustration.

Spline-based transformers are distinguished by the type of spline they use, i.e. by the functional form of the segments. The following options have been explored thus far, in order of increasing flexibility: linear and quadratic splines (Müller et al., 2019), cubic splines (Durkan et al., 2019a), linear-rational splines (Dolatabadi et al., 2020), and rational-quadratic splines (Durkan et al., 2019b). Spline-based transformers are as fast to invert as to evaluate, while maintaining exact analytical invertibility. Evaluating or inverting a spline-based transformer

is done by first locating the right segment—which can be done in $\mathcal{O}(\log K)$ time using binary search—and then evaluating or inverting that segment, which is assumed to be analytically tractable. By increasing the number of segments K, a spline-based transformer can be made arbitrarily flexible.

3.1.2 Implementing the Conditioner

The conditioner $c_i(\mathbf{z}_{<i})$ can be any function of $\mathbf{z}_{<i}$, meaning that each conditioner can, in principle, be implemented as an arbitrary model with input $\mathbf{z}_{<i}$ and output $\boldsymbol{h}_i$. However, a naïve implementation in which each $c_i(\mathbf{z}_{<i})$ is a separate model would scale poorly with the dimensionality D, requiring D model evaluations, each with a vector of average size $D/2$. This is in addition to the cost of storing and estimating the parameters of D independent models. In fact, early work on flow precursors (Chen and Gopinath, 2000) dismissed the autoregressive approach as prohibitively expensive.

Nonetheless, this problem can be effectively addressed in practice by sharing parameters across the conditioners $c_i(\mathbf{z}_{<i})$, or even by combining the conditioners into a single model. In the following paragraphs, we will discuss some practical implementations of the conditioner that allow it to scale to high dimensions.

Recurrent conditioners One way to share parameters across conditioners is by implementing them jointly using a *recurrent neural network* (RNN). The i-th conditioner is implemented as:

$$\boldsymbol{h}_i = c(\boldsymbol{s}_i) \quad \text{where} \quad \begin{array}{l} \boldsymbol{s}_1 = \text{initial state} \\ \boldsymbol{s}_i = \text{RNN}(\mathrm{z}_{i-1}, \boldsymbol{s}_{i-1}) \text{ for } i > 1. \end{array} \tag{39}$$

The RNN processes $\mathbf{z}_{<D} = (\mathrm{z}_1, \ldots, \mathrm{z}_{D-1})$ one element at a time, and at each step it updates a fixed-size internal state $\boldsymbol{s}_i$ that summarizes the subsequence $\mathbf{z}_{<i} = (\mathrm{z}_1, \ldots, \mathrm{z}_{i-1})$. The network c that computes $\boldsymbol{h}_i$ from $\boldsymbol{s}_i$ can be the same for each step. The initial state $\boldsymbol{s}_1$ can be fixed or it can be a learned parameter of the RNN. Any RNN architecture can be used, such as LSTM (Hochreiter and Schmidhuber, 1997) or GRU (Cho et al., 2014).

RNNs have been used extensively to share parameters across the conditional distributions of autoregressive models. Examples of RNN-based autoregressive models include distribution estimators (Larochelle and Murray, 2011; Uria et al., 2013, 2014), sequence models (Mikolov et al., 2010; Graves, 2013; Sutskever et al., 2014), and image/video models (Theis and Bethge, 2015; van den Oord et al., 2016b; Kalchbrenner et al., 2017). Section 3.1.3 discusses the relationship between autoregressive models and autoregressive flows in detail.

In the autoregressive-flows literature, RNN-based conditioners have been proposed by e.g. Oliva et al. (2018) and Kingma et al. (2016), but are relatively uncommon compared to alternatives. The main downside of RNN-based conditioners is that they turn an inherently parallel computation into a sequential one: the states $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_D$ must be computed sequentially, even though each $\boldsymbol{h}_i$ can in principle be computed independently and in parallel from $\mathbf{z}_{<i}$. Since this recurrent computation involves $\mathcal{O}(D)$ sequential steps, it can be slow for high-dimensional data such as images or videos.

Masked conditioners Another approach that shares parameters across conditioners but avoids the sequential computation of an RNN is based on *masking*. This approach uses a single, typically feedforward neural network that takes in $\mathbf{z}$ and outputs the entire sequence $(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_D)$ in one pass. The only requirement is that this network must obey the autoregressive structure of the conditioner: an output $\boldsymbol{h}_i$ cannot depend on inputs $\mathbf{z}_{\geq i}$.

To construct such a network, one takes an arbitrary neural network and removes connections until there is no path from input z_i to outputs $(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_i)$. A simple way to remove connections is by multiplying each weight matrix elementwise with a binary matrix of the same size. This has the effect of removing the connections corresponding to weights that are multiplied by zero, while leaving all other connections unmodified. These binary matrices can be thought of as 'masking out' connections, hence the term *masking*. The masked network will have the same architecture and size as the original network. In turn, it retains the computational properties of the original network, such as parallelism or ability to evaluate efficiently on a GPU.

A general procedure for constructing masks for multilayer perceptrons with arbitrarily many hidden layers or hidden units was proposed by Germain et al. (2015). The key idea is to assign a 'degree' between 1 and D to each input, hidden, and output unit, and mask-out the weights between subsequent layers such that no unit feeds into a unit with lower or equal degree. In convolutional networks, masking can be done by multiplying the filter with a binary matrix of the same size, which leads to a type of convolution often referred to as *autoregressive* or *causal* convolution (van den Oord et al., 2016c,a; Hoogeboom et al., 2019b). In architectures that use self-attention, masking can be done by zeroing out the softmax probabilities (Vaswani et al., 2017).

Masked autoregressive flows have two main advantages. First, they are efficient to evaluate. Given $\mathbf{z}$, the parameters $(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_D)$ can be obtained in one neural-network pass, and then each dimension of $\mathbf{z}'$ can be computed in parallel via $\mathrm{z}'_i = \tau(\mathrm{z}_i; \boldsymbol{h}_i)$. Second, masked autoregressive flows are universal approximators. Given a large enough conditioner and a flexible enough transformer, they can represent any autoregressive transformation with monotonic transformers and thus transform between any two distributions (as discussed in Section 2.2).

On the other hand, the main disadvantage of masked autoregressive flows is that they are not as efficient to invert as to evaluate. This is because the parameters $\boldsymbol{h}_i$ that are needed to obtain $\mathrm{z}_i = \tau^{-1}(\mathrm{z}'_i; \boldsymbol{h}_i)$ cannot be computed until all $(\mathrm{z}_i, \ldots, \mathrm{z}_{i-1})$ have been obtained. Following this logic, we must first compute $\boldsymbol{h}_1$ by which we obtain z_1, then compute $\boldsymbol{h}_2$ by which we obtain z_2, and so on until z_D has been obtained. Using a masked conditioner c, the above procedure can be implemented in pseudocode as follows:

$$\begin{aligned} &\text{Initialize } \mathbf{z} \text{ to an arbitrary value} \\ &\text{for } i = 1, \ldots, D \\ &\quad (\boldsymbol{h}_1, \ldots, \boldsymbol{h}_D) = c(\mathbf{z}) \\ &\quad \mathrm{z}_i = \tau^{-1}(\mathrm{z}'_i; \boldsymbol{h}_i). \end{aligned} \tag{40}$$

To see why this procedure is correct, observe that if $\mathbf{z}_{\leq i-1}$ is correct before the i-th iteration, then $\boldsymbol{h}_i$ will be computed correctly (due to the autoregressive structure of c) and thus $\mathbf{z}_{\leq i}$

will be correct before the $(i+1)$-th iteration. Since $\mathbf{z}_{\leq 0} = \varnothing$ is correct before the first iteration (in a degenerate sense, but still), by induction it follows that $\mathbf{z}_{\leq D} = \mathbf{z}$ will be correct at the end of the loop. Even though the above procedure can invert the flow exactly (provided the transformer is easy to invert), it requires calling the conditioner D times. This means that inverting a masked autoregressive flow using the above method is about D times more expensive than evaluating the forward transformation. For high-dimensional data such as images or video, this can be prohibitively expensive.

An alternative way to invert the flow, proposed by Song et al. (2019), is to solve the equation $\mathbf{z}' = f_{\boldsymbol{\phi}}(\mathbf{z})$ approximately, by iterating the following Newton-style fixed-point update:

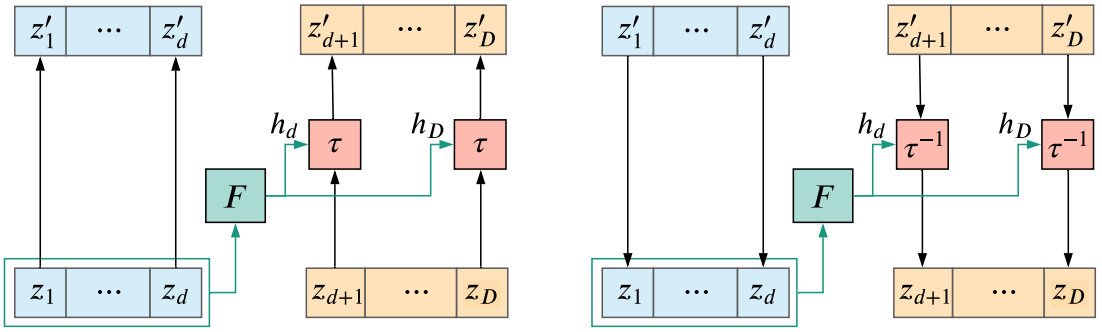
$$\mathbf{z}_{k+1} = \mathbf{z}_k - \alpha \operatorname{diag}\big(J_{f_{\boldsymbol{\phi}}}(\mathbf{z}_k)\big)^{-1}\big(f_{\boldsymbol{\phi}}(\mathbf{z}_k) - \mathbf{z}'\big), \tag{41}$$

where α is a step-size hyperparameter, and $\operatorname{diag}(\cdot)$ returns a diagonal matrix whose diagonal is the same as that of its input. A convenient initialization is $\mathbf{z}_0 = \mathbf{z}'$. Song et al. (2019) showed that the above procedure is locally convergent for $0 < \alpha < 2$, and since $f_{\boldsymbol{\phi}}^{-1}(\mathbf{z}')$ is the only fixed point, the procedure must either converge to it or diverge. With a masked autoregressive flow, computing both $f_{\boldsymbol{\phi}}(\mathbf{z}_k)$ and $\operatorname{diag}\big(J_{f_{\boldsymbol{\phi}}}(\mathbf{z}_k)\big)$ can be done efficiently by calling the conditioner once. Hence the above Newton-like procedure can be more efficient than inverting the flow exactly when the number of iterations to convergence in practice is significantly less than D. On the other hand, the above Newton-like procedure is approximate and guaranteed to converge only locally.

Despite the computational difficulties associated with inversion, masking remains one of the most popular techniques for implementing autoregressive flows. It is well suited to situations for which inverting the flow is not needed or the data dimensionality is not too large. Examples of flow-based models that use masking include IAF (Kingma et al., 2016), MAF (Papamakarios et al., 2017), NAF (Huang et al., 2018), block-NAF (De Cao et al., 2019), MintNet (Song et al., 2019) and MaCow (Ma et al., 2019). Masking can also be used and has been popular in implementing non-flow-based autoregressive models such as MADE (Germain et al., 2015), PixelCNN (van den Oord et al., 2016c; Salimans et al., 2017) and WaveNet (van den Oord et al., 2016a, 2018).

Coupling layers As we have seen, masked autoregressive flows have computational asymmetry that impacts their application and usability. Either sampling or density evaluation will be D times slower than the other. If both of these operations are required to be fast, a different implementation of the conditioner is needed. One such implementation that is computationally symmetric, i.e. equally fast to evaluate or invert, is the *coupling layer* (Dinh et al., 2015, 2017). The idea is to choose an index d (a common choice is $D/2$ rounded to an integer) and design the conditioner such that:

- Parameters $(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_d)$ are constants, i.e. not a function of $\mathbf{z}$.
- Parameters $(\boldsymbol{h}_{d+1}, \ldots, \boldsymbol{h}_D)$ are functions of $\mathbf{z}_{\leq d}$ only, i.e. they don't depend on $\mathbf{z}_{>d}$.



(a) Forward (b) Inverse

Figure 5: Illustration of a coupling layer.

This can be easily implemented using an arbitrary function approximator F (such as a neural network) as follows:

$$\begin{aligned}(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_d) &= \text{constants, either fixed or estimated} \\ (\boldsymbol{h}_{d+1}, \ldots, \boldsymbol{h}_D) &= F(\mathbf{z}_{\leq d}).\end{aligned} \tag{42}$$

In other words, the coupling layer splits $\mathbf{z}$ into two parts such that $\mathbf{z} = [\mathbf{z}_{\leq d}, \mathbf{z}_{>d}]$. The first part is transformed elementwise independently of other dimensions. The second part is transformed elementwise in a way that depends on the first part. We can also think of the coupling layer as implementing an aggressive masking strategy that allows only $(\boldsymbol{h}_{d+1}, \ldots, \boldsymbol{h}_D)$ to depend only on $\mathbf{z}_{\leq d}$.

Common implementations of coupling layers fix the transformers $\tau(\cdot; \boldsymbol{h}_1), \ldots, \tau(\cdot; \boldsymbol{h}_D)$ to the identity function. In this case, the transformation can be written as follows:

$$\begin{aligned}\mathbf{z}'_{\leq d} &= \mathbf{z}_{\leq d} \\ (\boldsymbol{h}_{d+1}, \ldots, \boldsymbol{h}_D) &= F(\mathbf{z}_{\leq d}) \\ \mathbf{z}'_i &= \tau(\mathbf{z}_i; \boldsymbol{h}_i) \text{ for } i > d.\end{aligned} \tag{43}$$

In turn, the inverse transformation is straightforward, given by:

$$\begin{aligned}\mathbf{z}_{\leq d} &= \mathbf{z}'_{\leq d} \\ (\boldsymbol{h}_{d+1}, \ldots, \boldsymbol{h}_D) &= F(\mathbf{z}_{\leq d}) \\ \mathbf{z}_i &= \tau^{-1}(\mathbf{z}'_i; \boldsymbol{h}_i) \text{ for } i > d.\end{aligned} \tag{44}$$

These are illustrated in Figure 5. Like all autoregressive flows, the Jacobian of the transformation is lower triangular, but in addition it has the following special structure:

$$J_{f_\phi} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{A} & \mathbf{D} \end{bmatrix}, \tag{45}$$

where $\mathbf{I}$ is the $d \times d$ identity matrix, $\mathbf{0}$ is the $d \times (D-d)$ zero matrix, $\mathbf{A}$ is a $(D-d) \times d$ full matrix, and $\mathbf{D}$ is a $(D-d) \times (D-d)$ diagonal matrix. The Jacobian determinant is

simply the product of the diagonal elements of $\mathbf{D}$, which are equal to the derivatives of the transformers $\tau(\cdot; \boldsymbol{h}_{d+1}), \ldots, \tau(\cdot; \boldsymbol{h}_D)$.

Coupling layers and fully autoregressive flows are two extremes on a spectrum of possible implementations. A coupling layer splits $\mathbf{z}$ into two parts and transforms the second part elementwise as a function of the first part, whereas a fully autoregressive flow splits the input into D parts (each with one element in it) and transforms each part as a function of all previous parts. Clearly, there are intermediate choices: one can split the input into K parts and transform the k-th part elementwise as a function of parts 1 to $k-1$, with $K = 2$ corresponding to a coupling layer and $K = D$ to a fully autoregressive flow. Using masking, inverting the transformation will be $\mathcal{O}(K)$ times more expensive than evaluating it, hence K could be chosen based on the computational requirements of the task.

The efficiency of coupling layers comes at the cost of reduced expressive power. Unlike a recurrent or masked autoregressive flow, a single coupling layer can no longer represent any autoregressive transformation, regardless of how expressive the function F is. As a result, an autoregressive flow with a single coupling layer is no longer a universal approximator. Nonetheless, the expressivity of the flow can be increased by composing multiple coupling layers. When composing coupling layers, the elements of $\mathbf{z}$ need to be permuted between layers so that all dimensions have a chance to be transformed as well as interact with one another. Previous work across various domains (see e.g. Kingma and Dhariwal, 2018; Prenger et al., 2019; Durkan et al., 2019b) has shown that composing coupling layers can indeed create flexible flows.

Theoretically, it is easy to show that a composition of D coupling layers is indeed a universal approximator, as long as the index d of the i-th coupling layer is equal to $i-1$. Observe that the i-th coupling layer can express any transformation of the form $\mathrm{z}'_i = \tau(\mathrm{z}_i; c_i(\mathbf{z}_{<i}))$, hence a composition of D such layers will have transformed each dimension fully autoregressively. However, this construction involves D sequential computations (one for each layer) in both the forward and inverse directions, so it doesn't provide an improvement over recurrent or masked autoregressive flows. It is an open problem whether it's possible to obtain a universal approximator by composing strictly fewer than $\mathcal{O}(D)$ coupling layers.

Coupling layers are one of the most popular methods for implementing flow-based models because they allow both density evaluation and sampling to be fast. A flow based on coupling layers can be tractably fitted with maximum likelihood and then be sampled from efficiently. Thus coupling layers are often found in generative models of high-dimensional data such as images, audio and video. Examples of flow-based models with coupling layers include NICE (Dinh et al., 2015), Real NVP (Dinh et al., 2017), Glow (Kingma and Dhariwal, 2018), WaveGlow (Prenger et al., 2019), FloWaveNet (Kim et al., 2019) and Flow++ (Ho et al., 2019).

3.1.3 Relationship with Autoregressive Models

Alongside normalizing flows, another popular class of models for high-dimensional distributions is the class of *autoregressive models*. Autoregressive models have a long history,

from the general framework of Bayesian networks (Pearl, 1988; Frey, 1998) to more recent neural-network-based implementations (Bengio and Bengio, 2000; Uria et al., 2016).

To construct an autoregressive model of $p_{\mathrm{x}}(\mathbf{x})$, we first decompose $p_{\mathrm{x}}(\mathbf{x})$ into a product of 1-dimensional conditionals using the chain rule of probability:

$$p_{\mathrm{x}}(\mathbf{x}) = \prod_{i=1}^{D} p_{\mathrm{x}}(\mathrm{x}_i \,|\, \mathbf{x}_{<i}). \tag{46}$$

We then model each conditional by some parametric distribution with parameters $\boldsymbol{h}_i$:

$$p_{\mathrm{x}}(\mathrm{x}_i \,|\, \mathbf{x}_{<i}) = p_{\mathrm{x}}(\mathrm{x}_i; \boldsymbol{h}_i), \quad \text{where} \quad \boldsymbol{h}_i = c_i(\mathbf{x}_{<i}). \tag{47}$$

For example, $p_{\mathrm{x}}(\mathrm{x}_i; \boldsymbol{h}_i)$ can be a Gaussian parameterized by its mean and variance, or a mixture of Gaussians parameterized by the mean, variance and mixture coefficient of each component. The functions c_i are analogous to the conditioners of an autoregressive flow, and are often implemented with neural networks using either RNNs or masking as discussed in previous sections. Apart from continuous data, autoregressive models can be readily used for discrete or even mixed data. If x_i is discrete for some i, then $p_{\mathrm{x}}(\mathrm{x}_i; \boldsymbol{h}_i)$ can be a parametric probability mass function such as a categorical or a mixture of Poissons.

We now show that *all* autoregressive models of continuous variables are in fact autoregressive flows with a single autoregressive layer. Let $\tau(\mathrm{x}_i; \boldsymbol{h}_i)$ be the cumulative distribution function of $p_{\mathrm{x}}(\mathrm{x}_i; \boldsymbol{h}_i)$, defined as follows:

$$\tau(\mathrm{x}_i; \boldsymbol{h}_i) = \int_{-\infty}^{\mathrm{x}_i} p_{\mathrm{x}}(\mathrm{x}'_i; \boldsymbol{h}_i)\, d\mathrm{x}'_i. \tag{48}$$

The function $\tau(\mathrm{x}_i; \boldsymbol{h}_i)$ is differentiable if $p_{\mathrm{x}}(\mathrm{x}_i; \boldsymbol{h}_i)$ is continuous, and strictly increasing if $p_{\mathrm{x}}(\mathrm{x}_i; \boldsymbol{h}_i) > 0$, both of which are the case in standard implementations of autoregressive models. As shown in Section 2.2, the vector $\mathbf{u} = (\mathrm{u}_1, \dots, \mathrm{u}_D)$, obtained by

$$\mathrm{u}_i = \tau(\mathrm{x}_i; \boldsymbol{h}_i) \quad \text{where} \quad \boldsymbol{h}_i = c_i(\mathbf{x}_{<i}), \tag{49}$$

is always distributed uniformly in $(0, 1)^D$. The above expression has exactly the same form as the definition of an autoregressive flow in Equation 29, with $\mathbf{z} = \mathbf{x}$ and $\mathbf{z}' = \mathbf{u}$. Therefore, an autoregressive model is in fact an autoregressive flow with a single autoregressive layer. Moreover, the layer's transformers are the cumulative distribution functions of the conditionals of the autoregressive model, and the layer's base distribution is a uniform in $(0, 1)^D$. We can make the connection explicit by writing the density under the change of variables:

$$\log p_{\mathrm{x}}(\mathbf{x}) = \log \prod_{i=1}^{D} \mathrm{Uniform}(\tau(\mathrm{x}_i; \boldsymbol{h}_i); 0, 1) + \log \prod_{i=1}^{D} p_{\mathrm{x}}(\mathrm{x}_i; \boldsymbol{h}_i) = \sum_{i=1}^{D} \log p_{\mathrm{x}}(\mathrm{x}_i \,|\, \mathbf{x}_{<i}). \tag{50}$$

The term involving the uniform base density drops from the expression, leaving just the Jacobian determinant.[2] Following Equation 30, the inverse autoregressive flow that maps

2. Inouye and Ravikumar (2018) termed flows of this form—whereby the density is fully determined by the Jacobian determinant—*density destructors*.

$\mathbf{u}$ to $\mathbf{x}$ is obtained by iterating the following for $i \in \{1, \ldots, D\}$:

$$\mathrm{z}_i = \tau^{-1}(\mathrm{u}_i; \boldsymbol{h}_i) \quad \text{where} \quad \boldsymbol{h}_i = c_i(\mathbf{x}_{<i}). \tag{51}$$

The above corresponds exactly to sampling from the autoregressive model one element at a time, where at each step the corresponding conditional is sampled from using *inverse transform sampling*.

Yet the transformer is not necessarily limited to being the inverse CDF. We can make further connections between specific types of autoregressive models and the transformers discussed in Section 3.1.1. For example, consider an autoregressive model with Gaussian conditionals of the form:

$$p_{\mathrm{x}}(\mathrm{x}_i; \boldsymbol{h}_i) = \mathcal{N}\left(\mathrm{x}_i; \mu_i, \sigma_i^2\right) \quad \text{where} \quad \boldsymbol{h}_i = \{\mu_i, \sigma_i\}. \tag{52}$$

The above conditional can be reparameterized as follows:

$$\mathrm{x}_i = \sigma_i \mathrm{u}_i + \mu_i \quad \text{where} \quad \mathrm{u}_i \sim \mathcal{N}(0, 1). \tag{53}$$

Hence, the entire autoregressive model can be reparameterized as an *affine autoregressive flow* as shown in Equation 33, where $\alpha_i = \sigma_i$, $\beta_i = \mu_i$, and the base distribution is a standard Gaussian (Kingma et al., 2016; Papamakarios et al., 2017). In the same way, we can relate other types of autoregressive models to non-affine transformers. For example, an autoregressive model whose conditionals are mixtures of Gaussians can be reparameterized as an autoregressive flow with combination-based transformers such as those in Equation 35. Similarly, an autoregressive model whose conditionals are histograms can be reparameterized as an autoregressive flow with transformers given by linear splines.

In consequence, we can think of autoregressive flows as subsuming and further extending autoregressive models for continuous variables. There are several benefits of viewing autoregressive models as flows. First, this view decouples the model architecture from the source of randomness, which gives us freedom in specifying the base distribution. Thus, we can enhance the flexibility of an autoregressive model by choosing a more flexible base distribution; for example, the base distribution can be another autoregressive model with its own learnable parameters. This provides a framework for composing autoregressive models, like layers in a flow (Papamakarios et al., 2017). Also, it allows us to compose autoregressive models with other types of flows, potentially non-autoregressive ones.

3.2 Linear Flows

As discussed in the previous section, autoregressive flows restrict an output variable z'_i to depend only on inputs $\mathbf{z}_{\leq i}$, making the flow dependent on the order of the input variables. As we showed, in the limit of infinite capacity, this restriction doesn't limit the flexibility of the flow-based model. However, in practice we don't operate at infinite capacity. The order of the input variables *will* determine the set of distributions the model can represent. Moreover, the target transformation may be easy to learn for some input orderings and hard to learn for others. The problem is further exacerbated when using coupling layers since only part of the input variables is transformed.

To cope with these limitations in practice, it often helps to permute the input variables between successive autoregressive layers. For coupling layers it is in fact necessary: if we don't permute the input variables between successive layers, part of the input will never be transformed. A permutation of the input variables is itself an easily invertible transformation, and its absolute Jacobian determinant is always 1 (i.e. it is volume-preserving). Hence, permutations can seamlessly be composed with other invertible and differentiable transformations the usual way.

An approach that generalizes the idea of a permutation of input variables is that of a *linear flow*. A linear flow is essentially an invertible linear transformation of the form:

$$\mathbf{z}' = \mathbf{W}\mathbf{z}, \tag{54}$$

where $\mathbf{W}$ is a $D \times D$ invertible matrix that parameterizes the transformation. The Jacobian of the above transformation is simply $\mathbf{W}$, making the Jacobian determinant equal to $\det \mathbf{W}$. A permutation is a special case of a linear flow, where $\mathbf{W}$ is a permutation matrix (i.e. a binary matrix with exactly one entry of 1 in each row and column and 0s everywhere else). Alternating invertible linear transformations with autoregressive/coupling layers is often used in practice (see e.g. Kingma and Dhariwal, 2018; Durkan et al., 2019b).

A straightforward implementation of a linear flow is to directly parameterize and learn the matrix $\mathbf{W}$. However, this approach can be problematic. First, $\mathbf{W}$ is not guaranteed to be invertible. Second, inverting the flow, which amounts to solving the linear system $\mathbf{W}\mathbf{z} = \mathbf{z}'$ for $\mathbf{z}$, takes time $\mathcal{O}(D^3)$ in general, which is prohibitively expensive for high-dimensional data. Third, computing $\det \mathbf{W}$ also takes time $\mathcal{O}(D^3)$ in general.

To address the above challenges, many approaches restrict $\mathbf{W}$ to a structured matrix, or a product of structured matrices. For example, if we restrict $\mathbf{W}$ to be triangular, we can guarantee its invertibility by e.g. making the diagonal elements positive. Moreover, inversion then costs $\mathcal{O}(D^2)$ (i.e. about the same as a matrix multiplication) and computing the determinant costs $\mathcal{O}(D)$. In Appendix B, we discuss in more detail this and a few more parameterizations that restrict the form of $\mathbf{W}$ in various ways.

In any case, it is important to note that it is *impossible* to parameterize all invertible matrices of size $D \times D$ in a continuous way, so any continuous parameterization of $\mathbf{W}$ that guarantees its invertibility will unavoidably leave out some invertible matrices. That's because there is no continuous surjective function from $\mathbb{R}^{D^2}$ to the set of $D \times D$ invertible matrices. To see why, consider two invertible matrices $\mathbf{W}_A$ and $\mathbf{W}_B$ such that $\det \mathbf{W}_A > 0$ and $\det \mathbf{W}_B < 0$. If there exists a continuous parameterization of all invertible matrices, then there exists a continuous path that connects $\mathbf{W}_A$ and $\mathbf{W}_B$. However, since the determinant is a continuous function of the matrix entries, any such path must include a matrix with zero determinant, i.e. a non-invertible matrix, which is a contradiction. This argument shows that the set of $D \times D$ invertible matrices contains two disconnected 'islands'—one containing matrices with positive determinant, the other with negative determinant—that are fully separated by the set of non-invertible matrices. In practice, this means that we can only hope to continuously parameterize one of these two islands, fixing the sign of the determinant from the outset.

3.3 Residual Flows

In this section, we consider a class of invertible transformations of the general form:

$$\mathbf{z}' = \mathbf{z} + g_{\boldsymbol{\phi}}(\mathbf{z}), \tag{55}$$

where $g_{\boldsymbol{\phi}}$ is a function that outputs a D-dimensional translation vector, parameterized by $\boldsymbol{\phi}$ (Figure 6). This structure bears a strong similarity to *residual networks* (He et al., 2016), and thus we use the term *residual flow* to refer to a normalizing flow composed of such transformations. Residual transformations are not always invertible, but can be made invertible if $g_{\boldsymbol{\phi}}$ is constrained appropriately. In what follows, we discuss two general approaches to designing invertible residual transformations: the first is based on *contractive maps*, and the second is based on the *matrix determinant lemma*.

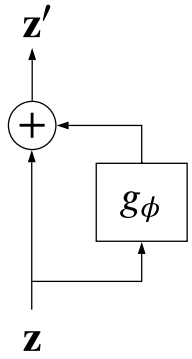


Figure 6: Residual flow

3.3.1 Contractive Residual Flows

A residual transformation is guaranteed to be invertible if $g_{\boldsymbol{\phi}}$ can be made *contractive* with respect to some distance function (Behrmann et al., 2019; Chen et al., 2019). In general, a map $F : \mathbb{R}^D \to \mathbb{R}^D$ is said to be contractive with respect to a distance function δ if there exists a constant $L < 1$ such that for any two inputs $\mathbf{z}_A$ and $\mathbf{z}_B$ we have:

$$\delta(F(\mathbf{z}_A), F(\mathbf{z}_B)) \leq L\, \delta(\mathbf{z}_A, \mathbf{z}_B). \tag{56}$$

In other words, a contractive map brings any two inputs closer together (as measured by δ) by at least a factor L. It directly follows that F is Lipschitz continuous with a Lipschitz constant equal to L. The *Banach fixed-point theorem* (Rudin, 1976, Theorem 9.23) states that any such contractive map has exactly one fixed point $\mathbf{z}_* = F(\mathbf{z}_*)$. Furthermore, this fixed point is the limit of any sequence $(\mathbf{z}_0, \mathbf{z}_1, \ldots)$ that is formed by an arbitrary starting point $\mathbf{z}_0$ and repeated application of F, i.e. $\mathbf{z}_{k+1} = F(\mathbf{z}_k)$ for all $k \geq 0$.

Invertibility of the residual transformation $\mathbf{z}' = f_{\boldsymbol{\phi}}(\mathbf{z}) = \mathbf{z} + g_{\boldsymbol{\phi}}(\mathbf{z})$ follows directly from $g_{\boldsymbol{\phi}}$ being contractive. Given $\mathbf{z}'$, consider the map:

$$F(\hat{\mathbf{z}}) = \mathbf{z}' - g_{\boldsymbol{\phi}}(\hat{\mathbf{z}}). \tag{57}$$

If $g_{\boldsymbol{\phi}}$ is contractive with Lipschitz constant L, then F is also contractive with the same Lipschitz constant. Hence, from the Banach fixed-point theorem, there exists a unique $\mathbf{z}_*$ such that $\mathbf{z}_* = \mathbf{z}' - g_{\boldsymbol{\phi}}(\mathbf{z}_*)$. By rearranging, we see that $\mathbf{z}' = f_{\boldsymbol{\phi}}(\mathbf{z}_*)$, and since $\mathbf{z}_*$ is unique, it follows that $f_{\boldsymbol{\phi}}$ is invertible.

In addition to a proof of the invertibility of $f_{\boldsymbol{\phi}}$, the above argument also gives us an algorithm for inversion. Starting from an arbitrary input $\mathbf{z}_0$ (a convenient choice is $\mathbf{z}_0 = \mathbf{z}'$), we can iteratively apply F as follows:

$$\mathbf{z}_{k+1} = \mathbf{z}' - g_{\boldsymbol{\phi}}(\mathbf{z}_k) \quad \text{for } k \geq 0. \tag{58}$$

The Banach fixed-point theorem guarantees that the above procedure converges to $\mathbf{z}_* = f_{\boldsymbol{\phi}}^{-1}(\mathbf{z}')$ for any choice of starting point $\mathbf{z}_0$. Moreover, it can be shown that the rate of convergence (with respect to δ) is exponential in the number of iterations k, and can be quantified as follows:

$$\delta(\mathbf{z}_k, \mathbf{z}_*) \leq \frac{L^k}{1-L}\,\delta(\mathbf{z}_0, \mathbf{z}_1). \tag{59}$$

The smaller the Lipschitz constant is, the faster $\mathbf{z}_k$ converges to $\mathbf{z}_*$. We can think of L as trading off flexibility for efficiency: as L gets smaller, the fewer iterations it takes to approximately invert the flow, but the residual transformation becomes more constrained, i.e. less flexible. In the extreme case of $L = 0$, the inversion procedure converges after one iteration, but the transformation reduces to adding a constant.

A challenge in building contractive residual flows is designing the function $g_{\boldsymbol{\phi}}$ to be contractive without impinging upon its flexibility. It is easy to see that the composition of K Lipschitz-continuous functions $F_1, \ldots, F_K$ is also Lipschitz continuous with a Lipschitz constant equal to $\prod_{k=1}^{K} L_k$, where L_k is the Lipschitz constant of F_k. Hence, if $g_{\boldsymbol{\phi}}$ is a composition of neural-network layers (as is common in deep learning), it is sufficient to make each layer Lipschitz continuous with $L_k \leq 1$, with at least one layer having $L_k < 1$, for the entire network to be contractive. Many elementwise nonlinearities used in deep learning—including the logistic sigmoid, hyperbolic tangent (tanh), and rectified linear (ReLU)—are in fact already Lipschitz continuous with a constant no greater than 1. Furthermore, linear layers (including dense layers and convolutional layers) can be made contractive with respect to a norm by dividing them with a constant strictly greater than their induced operator norm. One such implementation was proposed by Behrmann et al. (2019): spectral normalization (Miyato et al., 2018) was used to make linear layers contractive with respect to the Euclidean norm.

One drawback of contractive residual flows is that there is no known general, efficient procedure for computing their Jacobian determinant. Rather, one would have to revert to automatic differentiation to obtain the Jacobian and an explicit determinant computation to obtain the Jacobian determinant, which costs $\mathcal{O}(D^3)$ as discussed earlier. Without an efficient way to compute the Jacobian determinant, exactly evaluating the density of the flow model is costly and potentially infeasible for high-dimensional data such as images.

Nonetheless, it is possible to obtain an unbiased estimate of the log absolute Jacobian determinant, and hence of the log density, which is enough to train the flow model e.g. with maximum likelihood using stochastic gradients. We begin by writing the log absolute Jacobian determinant as a power series:[3]

$$\log\left|\det J_{f_{\boldsymbol{\phi}}}(\mathbf{z})\right| = \log\left|\det\big(\mathbf{I} + J_{g_{\boldsymbol{\phi}}}(\mathbf{z})\big)\right| = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} \operatorname{Tr}\Big\{J_{g_{\boldsymbol{\phi}}}^{k}(\mathbf{z})\Big\}, \tag{60}$$

where $J_{g_{\boldsymbol{\phi}}}^{k}(\mathbf{z})$ is the k-th power of the Jacobian of $g_{\boldsymbol{\phi}}$ evaluated at $\mathbf{z}$. The above series converges if $\left\|J_{g_{\boldsymbol{\phi}}}(\mathbf{z})\right\| < 1$ for some submultiplicative matrix norm $\|\cdot\|$, which in our case

3. This power series is essentially the Maclaurin series $\log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$ extended to matrices.

holds due to $g_{\boldsymbol{\phi}}$ being contractive. The trace of $J^k_{g_{\boldsymbol{\phi}}}(\mathbf{z})$ can be efficiently estimated using the *Hutchinson trace estimator* (Hutchinson, 1990):

$$\mathrm{Tr}\Big\{J^k_{g_{\boldsymbol{\phi}}}(\mathbf{z})\Big\} \approx \mathbf{v}^\top J^k_{g_{\boldsymbol{\phi}}}(\mathbf{z})\,\mathbf{v}, \tag{61}$$

where $\mathbf{v}$ can be any D-dimensional random vector with zero mean and unit covariance. The Jacobian-vector product $\mathbf{v}^\top J^k_{g_{\boldsymbol{\phi}}}(\mathbf{z})$ can then be computed with k backpropagation passes. Finally, the infinite sum can be estimated by a finite sum of appropriately re-weighted terms using the *Russian-roulette estimator* (Chen et al., 2019).

Unlike autoregressive flows, which are based on constraining the Jacobian to be sparse, contractive residual flows have a dense Jacobian in general, which allows all input variables to affect all output variables. As a result, contractive residual flows can be very flexible and have demonstrated good results in practice. On the other hand, unlike the one-pass density evaluation and sampling offered by flows based on coupling layers, exact density evaluation is computationally expensive and sampling is done iteratively, which limits the applicability of contractive residual flows in certain tasks.

3.3.2 Residual Flows Based on the Matrix Determinant Lemma

Suppose $\mathbf{A}$ is an invertible matrix of size $D \times D$ and $\mathbf{V}$, $\mathbf{W}$ are matrices of size $D \times M$. The *matrix determinant lemma* states:

$$\det\Big(\mathbf{A} + \mathbf{V}\mathbf{W}^\top\Big) = \det\Big(\mathbf{I} + \mathbf{W}^\top\mathbf{A}^{-1}\mathbf{V}\Big)\det\mathbf{A}. \tag{62}$$

If the determinant and inverse of $\mathbf{A}$ are tractable and M is less than D, the matrix determinant lemma can provide a computationally efficient way to compute the determinant of $\mathbf{A}+\mathbf{V}\mathbf{W}^\top$. For example, if $\mathbf{A}$ is diagonal, computing the left-hand side costs $\mathcal{O}\big(D^3 + D^2M\big)$, whereas computing the right-hand side costs $\mathcal{O}\big(M^3 + DM^2\big)$, which is preferable if $M < D$. In the context of flows, the matrix determinant lemma can be used to efficiently compute the Jacobian determinant. In this section, we will discuss examples of residual flows that are specifically designed such that application of the matrix determinant lemma leads to efficient Jacobian-determinant computation.

Planar flow One early example is the *planar flow* (Rezende and Mohamed, 2015), where the function $g_{\boldsymbol{\phi}}$ is a one-layer neural network with a single hidden unit:

$$\mathbf{z}' = \mathbf{z} + \mathbf{v}\sigma(\mathbf{w}^\top\mathbf{z} + b). \tag{63}$$

The parameters of the planar flow are $\mathbf{v} \in \mathbb{R}^D$, $\mathbf{w} \in \mathbb{R}^D$ and $b \in \mathbb{R}$, and σ is a differentiable activation function such as the hyperbolic tangent. This flow can be interpreted as expanding/contracting the space in the direction perpendicular to the hyperplane $\mathbf{w}^\top\mathbf{z} + b = 0$. The Jacobian of the transformation is given by:

$$J_{f_{\boldsymbol{\phi}}}(\mathbf{z}) = \mathbf{I} + \sigma'(\mathbf{w}^\top\mathbf{z} + b)\,\mathbf{v}\mathbf{w}^\top, \tag{64}$$

where σ' is the derivative of the activation function. The Jacobian has the form of a diagonal matrix plus a rank-1 update. Using the matrix determinant lemma, the Jacobian determinant can be computed in time $\mathcal{O}(D)$ as follows:

$$\det J_{f_\phi}(\mathbf{z}) = 1 + \sigma'(\mathbf{w}^\top \mathbf{z} + b)\, \mathbf{w}^\top \mathbf{v}. \tag{65}$$

In general, the planar flow is not invertible for all values of $\mathbf{v}$ and $\mathbf{w}$. However, assuming that σ' is positive everywhere and bounded from above (which is the case if σ is the hyperbolic tangent, for example), a sufficient condition for invertibility is $\mathbf{w}^\top \mathbf{v} > -\frac{1}{\sup_x \sigma'(x)}$.

Sylvester flow Planar flows can be extended to M hidden units, in which case they are known as *Sylvester flows* (van den Berg et al., 2018) and can be written as:

$$\mathbf{z}' = \mathbf{z} + \mathbf{V}\sigma(\mathbf{W}^\top \mathbf{z} + \mathbf{b}). \tag{66}$$

The parameters of the flow are now $\mathbf{V} \in \mathbb{R}^{D\times M}$, $\mathbf{W} \in \mathbb{R}^{D\times M}$ and $\mathbf{b} \in \mathbb{R}^M$, and the activation function σ is understood elementwise. The Jacobian can be written as:

$$J_{f_\phi}(\mathbf{z}) = \mathbf{I} + \mathbf{V}\mathbf{S}(\mathbf{z})\mathbf{W}^\top, \tag{67}$$

where $\mathbf{S}(\mathbf{z})$ is an $M \times M$ diagonal matrix whose diagonal is equal to $\sigma'(\mathbf{W}^\top \mathbf{z} + \mathbf{b})$. Applying the matrix determinant lemma we get:

$$\det J_{f_\phi}(\mathbf{z}) = \det\Big(\mathbf{I} + \mathbf{S}(\mathbf{z})\mathbf{W}^\top \mathbf{V}\Big), \tag{68}$$

which can be computed in $\mathcal{O}\big(M^3 + DM^2\big)$. To further reduce the computational cost, van den Berg et al. (2018) proposed the parameterization $\mathbf{V} = \mathbf{Q}\mathbf{U}$ and $\mathbf{W} = \mathbf{Q}\mathbf{L}$, where $\mathbf{Q}$ is a $D \times M$ matrix whose columns are an orthonormal set of vectors (this requires $M \leq D$), $\mathbf{U}$ is $M \times M$ upper triangular, and $\mathbf{L}$ is $M \times M$ lower triangular. Since $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}$ and the product of upper-triangular matrices is also upper triangular, the Jacobian determinant becomes:

$$\det J_{f_\phi}(\mathbf{z}) = \det\Big(\mathbf{I} + \mathbf{S}(\mathbf{z})\mathbf{L}^\top \mathbf{U}\Big) = \prod_{i=1}^{D}(1 + S_{ii}(\mathbf{z})L_{ii}U_{ii}). \tag{69}$$

Similar to planar flows, Sylvester flows are not invertible for all values of their parameters. Assuming σ' is positive everywhere and bounded from above, a sufficient condition for invertibility is $L_{ii}U_{ii} > -\frac{1}{\sup_x \sigma'(x)}$ for all $i \in \{1, \dots, D\}$.

Radial flow *Radial flows* (Tabak and Turner, 2013; Rezende and Mohamed, 2015) take the following form:

$$\mathbf{z}' = \mathbf{z} + \frac{\beta}{\alpha + r(\mathbf{z})}(\mathbf{z} - \mathbf{z}_0) \quad \text{where} \quad r(\mathbf{z}) = \|\mathbf{z} - \mathbf{z}_0\|. \tag{70}$$

The parameters of the flow are $\alpha \in (0, +\infty)$, $\beta \in \mathbb{R}$ and $\mathbf{z}_0 \in \mathbb{R}^D$, and $\|\cdot\|$ is the Euclidean norm. The above transformation can be thought of as a contraction/expansion radially with center $\mathbf{z}_0$. The Jacobian can be written as follows:

$$J_{f_\phi}(\mathbf{z}) = \left(1 + \frac{\beta}{\alpha + r(\mathbf{z})}\right)\mathbf{I} - \frac{\beta}{r(\mathbf{z})(\alpha + r(\mathbf{z}))^2}(\mathbf{z} - \mathbf{z}_0)(\mathbf{z} - \mathbf{z}_0)^\top, \tag{71}$$

which is a diagonal matrix plus a rank-1 update. Applying the matrix determinant lemma and rearranging, we get the following expression for the Jacobian determinant, which can be computed in $\mathcal{O}(D)$:

$$\det J_{f_\phi}(\mathbf{z}) = \left(1 + \frac{\alpha\beta}{(\alpha + r(\mathbf{z}))^2}\right)\left(1 + \frac{\beta}{\alpha + r(\mathbf{z})}\right)^{D-1}. \tag{72}$$

The radial flow is not invertible for all values of β. A sufficient condition for invertibility is $\beta > -\alpha$.

In summary, planar, Sylvester and radial flows have Jacobian determinants that cost $\mathcal{O}(D)$ to compute, and can be made invertible by suitably restricting their parameters. However, there is no analytical way of computing their inverse, which is why these flows have mostly been used to approximate posteriors for variational autoencoders. Moreover, each individual transformation is fairly simple, and it's not clear how the flexibility of the flow can be increased other than by increasing the number of transformations.

3.4 Practical Considerations when Combining Transformations

Implementing a flow often amounts to composing as many transformations as computation and memory will allow. For instance, Kingma and Dhariwal (2018)'s Glow architecture employs as many as 320 sub-transformations distributed across 40 GPUs to achieve state-of-the-art image generation. Working with such deep flows introduces additional challenges of a practical nature. In this section, we summarize two techniques that, respectively, stabilize the optimization and ease the computational demands of deep flows.

Normalization Like with deep neural networks trained with gradient-based methods, normalizing the intermediate representations $\mathbf{z}_k$ is crucial for maintaining stable gradients throughout the flow. *Batch normalization* or *batch norm* (Ioffe and Szegedy, 2015) has been widely demonstrated to be effective in stabilizing and improving neural-network training, thus making it attractive for use in deep flows as well. Viewing the batch statistics as fixed, batch norm is essentially a composition of two affine transformations. The first has scale and translation parameters set by the batch statistics, and the second has free parameters $\boldsymbol{\alpha}$ (scale) and $\boldsymbol{\beta}$ (translation):

$$\mathrm{BN}(\mathbf{z}) = \boldsymbol{\alpha} \odot \frac{\mathbf{z} - \hat{\boldsymbol{\mu}}}{\sqrt{\hat{\boldsymbol{\sigma}}^2 + \boldsymbol{\epsilon}}} + \boldsymbol{\beta}, \qquad \mathrm{BN}^{-1}(\mathbf{z}') = \hat{\boldsymbol{\mu}} + \frac{\mathbf{z}' - \boldsymbol{\beta}}{\boldsymbol{\alpha}} \odot \sqrt{\hat{\boldsymbol{\sigma}}^2 + \boldsymbol{\epsilon}}. \tag{73}$$

Moreover, batch norm has an easy-to-compute Jacobian determinant due to it acting elementwise (and thus having a diagonal Jacobian):

$$\det J_{\mathrm{BN}}(\mathbf{z}) = \prod_{i=1}^{D} \frac{\alpha_i}{\sqrt{\hat{\sigma}_i^2 + \epsilon_i}}. \tag{74}$$

In consequence, batch norm can be inserted between consecutive sub-transformations and treated simply as another member of the composition: $T_k \circ \mathrm{BN} \circ T_{k-1}$.

The above formulas assume that batch statistics are fixed, which is true for a trained model. During training, however, batch statistics are not fixed, but are functions of all examples in the batch. This makes batch norm not invertible, unless the batch statistics have been cached during a forward pass. Also, the Jacobian determinant as written above makes little sense mathematically, since batch norm is now a function of the whole batch. Yet, using this Jacobian-determinant formula as an approximation often suffices for training, at least if the batch is large enough (Dinh et al., 2017; Papamakarios et al., 2017).

Glow employs a variant termed *activation normalization* or *act norm* (Kingma and Dhariwal, 2018) that doesn't use batch statistics $\hat{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\sigma}}$. Instead, before training begins, a batch is passed through the flow, and $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are set such that the transformed batch has zero mean and unit variance. After this data-dependent initialization, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are optimized as model parameters. Act norm is preferable when training with small mini-batches since batch norm's statistics become noisy and can destabilize training.

Multi-scale architectures As mentioned in Section 2.1, $\mathbf{x}$ and $\mathbf{u}$ must have the same dimensionality and every sub-transformation T_k must preserve dimensionality. This means that evaluating T incurs an increasing computational cost as dimensionality grows. This constraint is at direct odds with our desire to use as many steps in the flow as possible. Dinh et al. (2017) proposed side-stepping this issue by way of a *multi-scale architecture*. At regular intervals in the steps of the flow when going from $\mathbf{x}$ to $\mathbf{u}$, some number of sub-dimensions of $\mathbf{z}_k$ are clamped and no additional transformation is applied. One can think of this as implementing a skip-connection, mapping those dimensions directly to the corresponding dimensions in the final representation $\mathbf{u}$: $(\mathrm{u}_j, \ldots, \mathrm{u}_i) = (\mathrm{z}_{k,j}, \ldots, \mathrm{z}_{k,i})$ where k is the step at which the clamping is applied. All K steps are applied to only a small subset of dimensions, which is less costly than applying all steps to all dimensions. Dinh et al. (2017) also argue that these skip-connections help with optimization, distributing the objective throughout the full depth of the flow.

Besides having this practical benefit, multi-scale architectures are a natural modeling choice for granular data types such as pixels (Dinh et al., 2017; Kingma and Dhariwal, 2018) and waveforms (Prenger et al., 2019; Kim et al., 2019). The macro-structures that we often care about—such as shapes and textures, in the case of images—typically do not need all D dimensions to be described. Dinh et al. (2017) showed that multi-scale architectures do indeed encode more global, semantic information in the dimensions that undergo all transformations. On the other hand, dimensions that are factored out earlier in the flow represent lower-level information; see Dinh et al. (2017)'s Appendix D for demonstrations.

4. Constructing Flows Part II: Continuous-Time Transformations

In the preceding section, we considered constructing flows by parameterizing a one-step transformation $\mathbf{z}_k = T_k(\mathbf{z}_{k-1})$, several of which are then composed to create a flow of K discrete steps. An alternative strategy is to construct flows in *continuous time* by parameterizing the flow's infinitesimal dynamics, and then *integrating* to find the corresponding transformation. In other words, we construct the flow by defining an *ordinary differential*

equation (ODE) that describes the flow's evolution in time. We call these 'continuous-time' flows as they evolve according to a real-valued scalar variable analogous to the number of steps. We call this scalar 'time' as it determines how long the dynamics are run. In this section, we will describe this class of continuous-time flows and summarize numerical tools necessary for their implementation.

4.1 Definition

Let $\mathbf{z}_t$ denote the flow's state at time t (or 'step' t, thinking in the discrete setting). Time t is assumed to run continuously from t_0 to t_1, such that $\mathbf{z}_{t_0} = \mathbf{u}$ and $\mathbf{z}_{t_1} = \mathbf{x}$. A *continuous-time flow* is constructed by parameterizing the time derivative of $\mathbf{z}_t$ with a function $g_{\boldsymbol{\phi}}$ with parameters $\boldsymbol{\phi}$, yielding the following *ordinary differential equation* (ODE):

$$\frac{d\mathbf{z}_t}{dt} = g_{\boldsymbol{\phi}}(t, \mathbf{z}_t). \tag{75}$$

The function $g_{\boldsymbol{\phi}}$ takes as inputs both the time t and the flow's state $\mathbf{z}_t$, and outputs the time derivative of $\mathbf{z}_t$ at time t. The only requirements for $g_{\boldsymbol{\phi}}$ are that it be uniformly Lipschitz continuous in $\mathbf{z}_t$ (meaning that there is a single Lipschitz constant that works for all t) and continuous in t (Chen et al., 2018). From *Picard's existence theorem*, it follows that satisfying these requirements ensures that the above ODE has a unique solution (Coddington and Levinson, 1955). Many neural-network layers meet these requirements (Gouk et al., 2018), and unlike the architectures described in Section 3 that require careful structural assumptions to ensure invertibility and tractability of their Jacobian determinant, $g_{\boldsymbol{\phi}}$ has no such requirements.

To compute the transformation $\mathbf{x} = T(\mathbf{u})$, we need to run the dynamics forward in time by integrating:

$$\mathbf{x} = \mathbf{z}_{t_1} = \mathbf{u} + \int_{t=t_0}^{t_1} g_{\boldsymbol{\phi}}(t, \mathbf{z}_t)\, dt. \tag{76}$$

The inverse transform T^{-1} is then:

$$\mathbf{u} = \mathbf{z}_{t_0} = \mathbf{x} + \int_{t=t_1}^{t_0} g_{\boldsymbol{\phi}}(t, \mathbf{z}_t)\, dt = \mathbf{x} - \int_{t=t_0}^{t_1} g_{\boldsymbol{\phi}}(t, \mathbf{z}_t)\, dt, \tag{77}$$

where in the right-most expression we used the fact that switching the limits of integration is equivalent to negating the integral. We write the inverse in this last form to show that, unlike many flows comprised of discrete compositions (Section 3), continuous-time flows *have the same computational complexity in each direction.* In consequence, choosing which direction is the forward and which is the inverse is not a crucial implementation choice as it is for e.g. autoregressive flows.

The change in log density for continuous-time flows can be characterized directly as (Chen et al., 2018):

$$\frac{d \log p(\mathbf{z}_t)}{dt} = -\mathrm{Tr}\Big\{ J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t) \Big\} \tag{78}$$

where $\mathrm{Tr}\{\cdot\}$ denotes the trace operator and $J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t)$ is the Jacobian of $g_{\boldsymbol{\phi}}(t, \cdot)$ evaluated at $\mathbf{z}_t$. The above equation can be obtained as a special case of the *Fokker–Planck equation* for

zero diffusion (Risken, 1996). While the trace operator at first glance seems more computationally tractable than a determinant, in practice it still requires $\mathcal{O}(D)$ backpropagation passes to obtain the diagonal elements of $J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t)$. Similarly to contractive residual flows (Section 3.3.1), the *Hutchinson's trace estimator* (Hutchinson, 1990) can be used to obtain an approximation in high-dimensional settings (Grathwohl et al., 2019):

$$\mathrm{Tr}\Big\{J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t)\Big\} \approx \mathbf{v}^\top J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t)\,\mathbf{v}, \tag{79}$$

where $\mathbf{v}$ can be any D-dimensional random vector with zero mean and unit covariance. The Jacobian-vector product $\mathbf{v}^\top J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t)$ can be computed in a single backpropagation pass, which makes the Hutchinson trace estimator about D times more efficient than calculating the trace exactly. Chen and Duvenaud (2019) propose an alternative solution in which the architecture of $g_{\boldsymbol{\phi}}$ is carefully constrained so that the exact Jacobian trace can be computed in a single backpropagation pass.

By integrating the derivative of $\log p(\mathbf{z}_t)$ over time, we obtain an expression for the log density of $\mathbf{x}$ under the continuous-time flow:

$$\log p_{\mathrm{x}}(\mathbf{x}) = \log p_{\mathrm{u}}(\mathbf{u}) - \int_{t=t_0}^{t_1} \mathrm{Tr}\Big\{J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t)\Big\}\,dt. \tag{80}$$

Evaluating the forward transform and the log density can be done simultaneously by computing the following combined integral:

$$\begin{bmatrix}\mathbf{x}\\ \log p_{\mathrm{x}}(\mathbf{x})\end{bmatrix} = \begin{bmatrix}\mathbf{u}\\ \log p_{\mathrm{u}}(\mathbf{u})\end{bmatrix} + \int_{t=t_0}^{t_1} \begin{bmatrix} g_{\boldsymbol{\phi}}(t,\mathbf{z}_t)\\ -\mathrm{Tr}\Big\{J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t)\Big\}\end{bmatrix} dt. \tag{81}$$

Computation is not analytically feasible for a general $g_{\boldsymbol{\phi}}$. In practice, a numerical integrator is used, as we discuss next.

4.2 Solving and Optimizing Continuous-Time Flows

Due to continuous-time flows being defined by an ODE, the vast literature on numerical ODE solvers and the corresponding software can be leveraged to implement these flows. See Süli (2010) for an accessible introduction to numerical methods for ODEs. While there are numerous numerical methods that could be employed, below we briefly describe *Euler's method* and the *adjoint method*.

4.2.1 Euler's Method and Equivalence to Residual Flows

Perhaps the simplest numerical technique one can apply is *Euler's method*. The idea is to first discretize the ODE using a small step-size $\epsilon > 0$ as follows:

$$\mathbf{z}_{t+\epsilon} \approx f_{\boldsymbol{\phi}}(\mathbf{z}_t) = \mathbf{z}_t + \epsilon\, g_{\boldsymbol{\phi}}(t,\mathbf{z}_t) \tag{82}$$

with the approximation becoming exact as $\epsilon \to 0$. The ODE can then be (approximately) solved by iterating the above computation starting from $\mathbf{z}_{t_0} = \mathbf{u}$ until obtaining $\mathbf{z}_{t_1} = \mathbf{x}$.

This way, the parameters $\boldsymbol{\phi}$ can be optimized with gradients computed via backpropagation through the ODE solver. It is relatively straightforward to use other discrete solvers such as any in the *Runge–Kutta* family. The discretized forward solution would be backpropagated through just as with Euler's method.

Interestingly, the Euler approximation implements the continuous-time flow as a discrete-time residual flow of the class described in Equation 55. Having assumed that $g_{\boldsymbol{\phi}}(t, \cdot)$ is uniformly Lipschitz continuous with a Lipschitz constant L independent of t, it immediately follows that $\epsilon\, g_{\boldsymbol{\phi}}(t, \cdot)$ is contractive for any $\epsilon < 1/L$. Hence, for small enough ϵ we can think of the above Euler discretization as a particular instance of a contractive residual flow (Section 3.3.1).

Approximating continuous-time flows using discrete-time residual flows gives us insight on Equation 78's description of the time evolution of $\log p(\mathbf{z}_t)$. Using the Taylor-series expansion of Equation 60, we can write the log absolute Jacobian determinant of $f_{\boldsymbol{\phi}}$ as follows:

$$\log \left|\det J_{f_{\boldsymbol{\phi}}}(\mathbf{z}_t)\right| = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}\epsilon^k}{k} \operatorname{Tr}\Big\{ J^k_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t) \Big\} = \epsilon \operatorname{Tr}\Big\{ J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t) \Big\} + \mathcal{O}\big(\epsilon^2\big). \tag{83}$$

Substituting the above into the change-of-variables formula and rearranging we get:

$$\log p(\mathbf{z}_{t+\epsilon}) = \log p(\mathbf{z}_t) - \epsilon \operatorname{Tr}\Big\{ J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t) \Big\} + \mathcal{O}\big(\epsilon^2\big) \Rightarrow \tag{84}$$

$$\frac{\log p(\mathbf{z}_{t+\epsilon}) - \log p(\mathbf{z}_t)}{\epsilon} = -\operatorname{Tr}\Big\{ J_{g_{\boldsymbol{\phi}}(t,\cdot)}(\mathbf{z}_t) \Big\} + \mathcal{O}(\epsilon), \tag{85}$$

from which we directly obtain Equation 78 by letting $\epsilon \to 0$.

4.2.2 The Adjoint Method

Chen et al. (2018) proposed an elegant alternative to the discrete, fixed-step methods mentioned above. For a general optimization target $\mathcal{L}(\mathbf{x}; \boldsymbol{\phi})$ (such as log likelihood), they show that the gradient $\partial\mathcal{L}/\partial\mathbf{z}_t$ with respect to the flow's intermediate state $\mathbf{z}_t$ can be characterized by the following ODE:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}_t}\right) = -\left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}_t}\right)^{\top} \frac{\partial g_{\boldsymbol{\phi}}(t, \mathbf{z}_t)}{\partial \mathbf{z}_t}, \tag{86}$$

a result known widely as the *adjoint sensitivity method* (Pontryagin, 1962). In neural-network terminology, the adjoint method can be thought of as the continuous-time analog of backpropagation. The gradient with respect to $\boldsymbol{\phi}$ can be computed by:

$$\frac{\partial \mathcal{L}(\mathbf{x}; \boldsymbol{\phi})}{\partial \boldsymbol{\phi}} = \int_{t=t_1}^{t_0} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_t} \frac{\partial g_{\boldsymbol{\phi}}(t, \mathbf{z}_t)}{\partial \boldsymbol{\phi}}\, dt. \tag{87}$$

Optimization of $\boldsymbol{\phi}$ can then be done with stochastic gradients.

Formulating gradient computation as a separate ODE means that both forward evaluation and gradient computation can be treated by black-box ODE solvers, without the need to

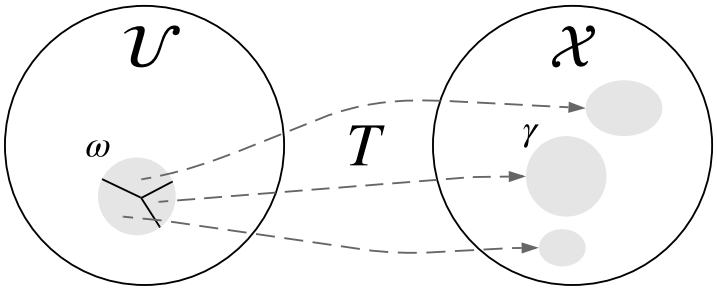


Figure 7: Illustration of the general probability-transformation formula. The probability $\Pr(\mathbf{u} \in \omega)$ must be equal to the probability $\Pr(\mathbf{x} \in \gamma)$ for any $\omega \subseteq \mathcal{U}$, and this relation will remain true even if the image of ω through T has disconnected components.

backpropagate through the solver's computational graph. This results in significant practical benefits since backpropagating through a solver is costly both in terms of computation and memory requirements. Another benefit is that a more sophisticated solver can allocate instance-dependent computation based on a user-specified tolerance, a common hyperparameter in most off-the-shelf solvers. At test time, this tolerance level can be tuned based on runtime or other constraints on computation.

5. Generalizations

So far, we have discussed normalizing flows as invertible transformations in Euclidean space. In this section, we go beyond the standard definition and explore more general classes of probability transformations. We show how more general types of flow can be derived from a unifying framework, point to specific implementations that extend the standard definition, and explore the frontiers of normalizing-flow research.

5.1 General Probability-Transformation Formula

Assume a probability density $p_{\mathrm{u}} : \mathcal{U} \to [0, +\infty)$ defined on a set $\mathcal{U}$, and a transformation $T : \mathcal{U} \to \mathcal{X}$ that maps the set $\mathcal{U}$ to the set $\mathcal{X}$. The density $p_{\mathrm{u}}(\mathbf{u})$ together with the transformation T induce a probability density $p_{\mathrm{x}} : \mathcal{X} \to [0, +\infty)$ on $\mathcal{X}$ via the general relation:

$$\int_{\mathbf{u} \in \omega \subseteq \mathcal{U}} p_{\mathrm{u}}(\mathbf{u})\, d\mu(\mathbf{u}) = \int_{\mathbf{x} \in \gamma \subseteq \mathcal{X}} p_{\mathrm{x}}(\mathbf{x})\, d\nu(\mathbf{x}) \quad \forall \omega \subseteq \mathcal{U}. \tag{88}$$

This is illustrated in Figure 7. In the above formula, $\gamma = \{T(\mathbf{u}) \,|\, \mathbf{u} \in \omega\}$ is the image of ω under T, and $d\mu(\mathbf{u})$ and $d\nu(\mathbf{x})$ are the integration measures in $\mathcal{U}$ and $\mathcal{X}$ with respect to which the densities $p_{\mathrm{u}}(\mathbf{u})$ and $p_{\mathrm{x}}(\mathbf{x})$ are defined. The formula has a simple interpretation as *conservation of probability measure*: its LHS is the probability that a sample from $p_{\mathrm{u}}(\mathbf{u})$

falls in $\omega \subseteq \mathcal{U}$, whereas the RHS is the probability that a sample from $p_{\mathrm{x}}(\mathbf{x})$ falls in $\gamma \subseteq \mathcal{X}$. Since γ is the image of ω under T, these two probabilities must be the same for any ω.

Standard flow-based models are a special case of this formula, where T is a diffeomorphism (i.e. an differentiable invertible transformation with differentiable inverse), the sets $\mathcal{U}$ and $\mathcal{X}$ are equal to the D-dimensional Euclidean space $\mathbb{R}^D$, and $d\mu(\mathbf{u})$ and $d\nu(\mathbf{x})$ are equal to the Lebesgue measure in $\mathbb{R}^D$. In that case, the conservation of probability measure can be written as:

$$\int_{\mathbf{u}\in\omega} p_{\mathrm{u}}(\mathbf{u})\, d\mathbf{u} = \int_{\mathbf{x}\in\gamma} p_{\mathrm{x}}(\mathbf{x})\, d\mathbf{x}. \tag{89}$$

Since T is a diffeomorphism, we can express the LHS integral via the change of variable $\mathbf{u} = T^{-1}(\mathbf{x})$ as follows (Rudin, 2006):

$$\int_{\mathbf{x}\in\gamma} p_{\mathrm{u}}\big(T^{-1}(\mathbf{x})\big)\, \left|\det J_{T^{-1}}(\mathbf{x})\right|\, d\mathbf{x} = \int_{\mathbf{x}\in\gamma} p_{\mathrm{x}}(\mathbf{x})\, d\mathbf{x}. \tag{90}$$

Since the above must be true for any $\gamma \subseteq \mathcal{X}$, it follows that:

$$p_{\mathrm{x}}(\mathbf{x}) = p_{\mathrm{u}}\big(T^{-1}(\mathbf{x})\big)\, \left|\det J_{T^{-1}}(\mathbf{x})\right|, \tag{91}$$

which is the familiar formula for the density of a flow-based model.

Using the conservation of probability measure as a general framework, we study different transformations T, sets $\mathcal{U}$ and $\mathcal{X}$, and integration measures $d\mu(\mathbf{u})$ and $d\nu(\mathbf{x})$. In the following sections, we explore further possibilities and potential implementations.

5.2 Piecewise-Invertible Transformations and Mixtures of Flows

We can extend the transformation T to be *many-to-one*, whereby multiple $\mathbf{u}$'s are mapped to the same $\mathbf{x}$. One possibility is for T to be *piecewise invertible*, in which case we can partition $\mathcal{U}$ into a countable collection of non-overlapping subsets $\{\mathcal{U}_i\}_{i\in\mathcal{I}}$ such that the restriction of T to $\mathcal{U}_i$ is an invertible transformation $T_i : \mathcal{U}_i \to \mathcal{X}$ (Figure 8a). Then, the conservation of probability measure can be written as:

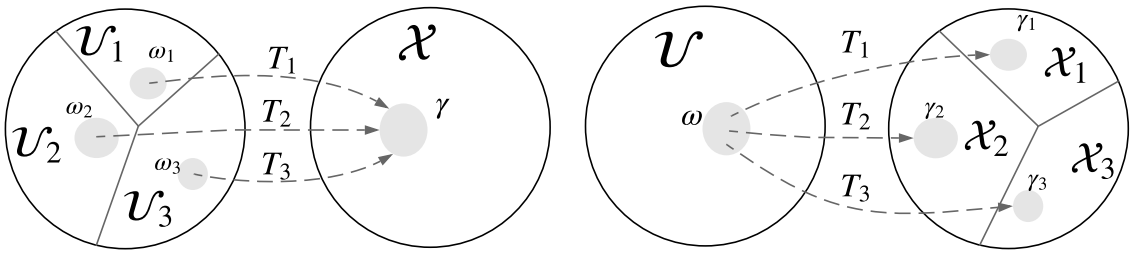
$$\sum_{i\in\mathcal{I}} \int_{\mathbf{u}\in\omega_i} p_{\mathrm{u}}(\mathbf{u})\, d\mathbf{u} = \int_{\mathbf{x}\in\gamma} p_{\mathrm{x}}(\mathbf{x})\, d\mathbf{x}, \tag{92}$$

where each $\omega_i \subseteq \mathcal{U}_i$ is the image of γ under T_i^{-1}. Using the change of variables $\mathbf{x} = T_i(\mathbf{u})$ for each corresponding integral on the LHS, we obtain:

$$\int_{\mathbf{x}\in\gamma} \sum_{i\in\mathcal{I}} p_{\mathrm{u}}\big(T_i^{-1}(\mathbf{x})\big)\, \left|\det J_{T_i^{-1}}(\mathbf{x})\right|\, d\mathbf{x} = \int_{\mathbf{x}\in\gamma} p_{\mathrm{x}}(\mathbf{x})\, d\mathbf{x}. \tag{93}$$

Since the above must be true for all $\gamma \subseteq \mathcal{X}$, it follows that:

$$p_{\mathrm{x}}(\mathbf{x}) = \sum_{i\in\mathcal{I}} p_{\mathrm{u}}\big(T_i^{-1}(\mathbf{x})\big)\, \left|\det J_{T_i^{-1}}(\mathbf{x})\right|. \tag{94}$$



(a) Many-to-one (b) One-to-many

Figure 8: Illustrations of a piecewise-invertible transformations.

The above can be interpreted as a *mixture of flows*, where the i-th flow has transformation T_i, base density $p_{\mathrm{u}}(\mathbf{u})$ restricted to $\mathcal{U}_i$ and mixture weight equal to $\Pr(\mathbf{u} \in \mathcal{U}_i)$.

An alternative would be to take the transformation T to be *one-to-many*, where a single $\mathbf{u}$ is mapped to multiple $\mathbf{x}$'s. One possibility, proposed by Dinh et al. (2019) and known as *real and discrete* (RAD), is to think of T as the 'inverse' of a piecewise-invertible transformation $R : \mathcal{X} \to \mathcal{U}$. We partition $\mathcal{X}$ into a countable collection of non-overlapping subsets $\{\mathcal{X}_i\}_{i\in\mathcal{I}}$ such that the restriction of R to $\mathcal{X}_i$ is an invertible transformation whose inverse is $T_i : \mathcal{U} \to \mathcal{X}_i$ (Figure 8b). Then, T simply maps $\mathbf{u}$ to $\{T_i(\mathbf{u})\}_{i\in\mathcal{I}}$. To make sense of the above as a generative model of $\mathbf{x}$, we can select one of $\{T_i(\mathbf{u})\}_{i\in\mathcal{I}}$ at random from some distribution $p(i \,|\, \mathbf{u})$ over $\mathcal{I}$ that can depend on $\mathbf{u}$. This is equivalent to extending the input space to $\mathcal{U}' = \mathcal{U} \times \mathcal{I}$, defining the input density $p(\mathbf{u}, i) = p_{\mathrm{u}}(\mathbf{u})\, p(i \,|\, \mathbf{u})$ with respect to the Lebesgue measure on $\mathcal{U}$ and the counting measure on $\mathcal{I}$, and defining the extended transformation $T' : \mathcal{U}' \to \mathcal{X}$ to be $T'(\mathbf{u}, i) = T_i(\mathbf{u})$. Then, the conservation of measure in the extended space can be written as:

$$\sum_{i\in\xi} \int_{\mathbf{u}\in\omega} p_{\mathrm{u}}(\mathbf{u})\, p(i \,|\, \mathbf{u})\, d\mathbf{u} = \sum_{i\in\xi} \int_{\mathbf{x}\in\gamma_i} p_{\mathrm{x}}(\mathbf{x})\, d\mathbf{x} \quad \forall \xi \subseteq \mathcal{I},\ \forall \omega \subseteq \mathcal{U}, \tag{95}$$

where $\gamma_i \subseteq \mathcal{X}_i$ is the image of ω under T_i. Using the change of variables $\mathbf{x} = T_i(\mathbf{u})$ for each corresponding integral on the LHS, we obtain:

$$\sum_{i\in\xi} \int_{\mathbf{x}\in\gamma_i} p_{\mathrm{u}}\big(T_i^{-1}(\mathbf{x})\big)\, p\big(i \,|\, T_i^{-1}(\mathbf{x})\big)\, \Big|\det J_{T_i^{-1}}(\mathbf{x})\Big|\, d\mathbf{x} = \sum_{i\in\xi} \int_{\mathbf{x}\in\gamma_i} p_{\mathrm{x}}(\mathbf{x})\, d\mathbf{x}. \tag{96}$$

The above must be true for all $\xi \subseteq \mathcal{I}$ and $\gamma_i \subseteq \mathcal{X}$, and by definition $T_i^{-1}(\mathbf{x}) = R(\mathbf{x})$ for all $\mathbf{x} \in \mathcal{X}_i$, therefore:

$$p_{\mathrm{x}}(\mathbf{x}) = p_{\mathrm{u}}(R(\mathbf{x}))\, p(i(\mathbf{x}) \,|\, R(\mathbf{x}))\, |\det J_R(\mathbf{x})|\,, \tag{97}$$

where $i(\mathbf{x})$ indexes the subset of $\mathcal{X}$ in which $\mathbf{x}$ belongs. The above can be interpreted as a mixture of flows with non-overlapping components, where the i-th component uses transformation $T_i : \mathcal{U} \to \mathcal{X}_i$, base distribution $p_{\mathrm{u}}(\mathbf{u} \,|\, i) \propto p_{\mathrm{u}}(\mathbf{u})\, p(i \,|\, \mathbf{u})$ and mixture weight $p(i) = \int p_{\mathrm{u}}(\mathbf{u})\, p(i \,|\, \mathbf{u})\, d\mathbf{u}$.

5.3 Flows for Discrete Random Variables

Using the general probability-transformation formula, we can extend normalizing flows to discrete sets $\mathcal{U}$ and $\mathcal{X}$. Taking the integration measures $d\mu(\mathbf{u})$ and $d\nu(\mathbf{x})$ to be the counting measure, we can write the conservation of measure as follows:

$$\sum_{\mathbf{u}\in\omega} p_{\mathrm{u}}(\mathbf{u}) = \sum_{\mathbf{x}\in\gamma} p_{\mathrm{x}}(\mathbf{x}). \tag{98}$$

Letting $\gamma = \{\mathbf{x}\}$ for an arbitrary $\mathbf{x}$, we obtain:

$$p_{\mathrm{x}}(\mathbf{x}) = \sum_{\mathbf{u}\in\omega} p_{\mathrm{u}}(\mathbf{u}) \quad \text{where} \quad \omega = \{\mathbf{u} \,|\, T(\mathbf{u}) = \mathbf{x}\}\,. \tag{99}$$

Finally, by restricting T to be bijective (assuming $\mathcal{U}$ and $\mathcal{X}$ have the same cardinality), we obtain a normalizing flow on discrete random variables, whose density[4] is given by:

$$p_{\mathrm{x}}(\mathbf{x}) = p_{\mathrm{u}}\big(T^{-1}(\mathbf{x})\big). \tag{100}$$

We will refer to the above type of normalizing flow as a *discrete flow*. Unlike standard flows, discrete flows don't involve a Jacobian term in their density calculation.

Hoogeboom et al. (2019a) proposed a discrete flow for $\mathcal{U} = \mathcal{X} = \mathbb{Z}^D$ based on affine autoregressive flows (Section 3.1). Specifically, they implement the transformer $\tau : \mathbb{Z} \to \mathbb{Z}$ as follows:

$$\tau(\mathrm{z}_i; \beta_i) = \mathrm{z}_i + \mathrm{round}(\beta_i), \tag{101}$$

where β_i is given by the conditioner and is a function of $\mathbf{z}_{<i}$, and round$(\cdot)$ maps its input to the nearest integer. Similarly, Tran et al. (2019) proposed a discrete flow for $\mathcal{U} = \mathcal{X} = \{0, \dots, K-1\}^D$ also based on affine autoregressive flows and whose transformer $\tau : \{0, \dots, K-1\} \to \{0, \dots, K-1\}$ is given by:

$$\tau(\mathrm{z}_i; \alpha_i, \beta_i) = (\alpha_i \mathrm{z}_i + \beta_i) \bmod K, \tag{102}$$

where α_i and β_i are each given by the argmax of a K-dimensional vector outputted by the conditioner. The above transformer can be shown to be bijective whenever α_i and K are coprime. To backpropagate through the discrete-valued functions round$(\cdot)$ and argmax, both Hoogeboom et al. (2019a) and Tran et al. (2019) use the straight-through gradient estimator (Bengio et al., 2013).

Compared to flows on $\mathbb{R}^D$, discrete flows have notable theoretical limitations. As shown in Section 2.2, under mild conditions, flows on $\mathbb{R}^D$ can transform any base density $p_{\mathrm{u}}(\mathbf{u})$ to any target density $p_{\mathrm{x}}(\mathbf{x})$. However, this is not true for discrete flows—at least not as long as they are defined using a bijective transformation. For example, if the base distribution $p_{\mathrm{u}}(\mathbf{u})$ of a discrete flow is uniform, $p_{\mathrm{x}}(\mathbf{x})$ will necessarily be uniform too. More generally, due to bijectivity, for every $\mathbf{x}$ there must be a $\mathbf{u}$ such that $p_{\mathrm{x}}(\mathbf{x}) = p_{\mathrm{u}}(\mathbf{u})$. In other words, a discrete flow can never change the values of $p_{\mathrm{u}}(\mathbf{u})$, only permute them. Recalling the change-of-variables formula for discrete flows (Equation 100), the absence of a Jacobian term may

4. The probability mass function $p_{\mathrm{x}}(\mathbf{x})$ can be though of as a density with respect to the counting measure.

seem to be a computational boon that makes discrete flows preferable to continuous ones. However, this absence is also a restriction.

Another useful property of standard flows is that they can model any target density using a fully factorized base distribution $p_{\mathrm{u}}(\mathbf{u}) = \prod_{i=1}^{D} p_{\mathrm{u}}(\mathrm{u}_i)$. Such a base density can be evaluated and sampled from in parallel, which is important for scalability to high dimensions. Nonetheless, a discrete flow with a fully factorized base distribution may not be able to model all target densities, even if the base distribution is learned. To see this, consider the target density $p_{\mathrm{x}}(\mathrm{x}_1, \mathrm{x}_2)$ with $\mathrm{x}_1 \in \{0, 1\}$, $\mathrm{x}_2 \in \{0, 1\}$ given by:

$$\begin{aligned} p_{\mathrm{x}}(0,0) &= 0.1 \qquad p_{\mathrm{x}}(0,1) = 0.3 \\ p_{\mathrm{x}}(1,0) &= 0.2 \qquad p_{\mathrm{x}}(1,1) = 0.4. \end{aligned} \tag{103}$$

Assuming $\mathcal{U} = \mathcal{X} = \{0, 1\}^2$, all a discrete flow can do to $p_{\mathrm{x}}(\mathrm{x}_1, \mathrm{x}_2)$ is permute the 4 values of the probability table:

$$\begin{bmatrix} 0.1 & 0.3 \\ 0.2 & 0.4 \end{bmatrix}$$

into a new 2×2 probability table. Therefore, to model $p_{\mathrm{x}}(\mathrm{x}_1, \mathrm{x}_2)$ with a factorized base $p_{\mathrm{u}}(\mathrm{u}_1, \mathrm{u}_2) = p_{\mathrm{u}}(\mathrm{u}_1) p_{\mathrm{u}}(\mathrm{u}_2)$, there must be a permutation of the above table such that the permuted table is of rank 1 (so it can be factorized as the outer product of two vectors). However, after checking all $4! = 24$ permutations, we find that all permuted tables are of rank 2, which shows that the above target can't be modeled using a factorized base. This limitation means that, in practice, we may need to explicitly incorporate dependencies in the base distribution to increase the model's capacity. Both Hoogeboom et al. (2019a) and Tran et al. (2019) take this approach, modeling the base distribution autoregressively.

A possible way to overcome this limitation, pointed out by van den Berg et al. (2020), is to embed $\mathcal{U}$ and $\mathcal{X}$ into extended spaces $\mathcal{U}'$ and $\mathcal{X}'$, such that the base distribution factorizes in the extended space. In the above example, we could take $\mathcal{U}' = \mathcal{X}' = \{0, 1, 2, 3\}^2$, so that the probability table becomes:

$$\begin{bmatrix} 0.1 & 0.3 & 0 & 0 \\ 0.2 & 0.4 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Then, a discrete flow on the extended space can rearrange the probability table into:

$$\begin{bmatrix} 0.1 & 0 & 0 & 0 \\ 0.2 & 0 & 0 & 0 \\ 0.3 & 0 & 0 & 0 \\ 0.4 & 0 & 0 & 0 \end{bmatrix},$$

which is rank 1 and thus can be factorized.

5.4 Flows on Riemannian Manifolds

Next, we can extend normalizing flows to Riemannian manifolds embedded in a higher-dimensional space (Gemici et al., 2016; Wang and Wang, 2019). Assume that $\mathcal{X}$ is a

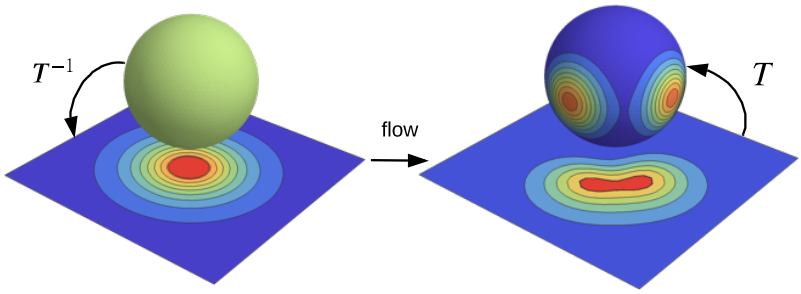


Figure 9: Density on a 2-dimensional sphere formed by mapping the sphere to $\mathbb{R}^2$, transforming the density there, and mapping $\mathbb{R}^2$ back to the sphere.

D-dimensional manifold embedded in $\mathbb{R}^M$ via an injective map $T : \mathbb{R}^D \to \mathbb{R}^M$ with $M \geq D$. The embedding map T can be thought of as a curvilinear coordinate system that traces points $\mathbf{x} \in \mathbb{R}^M$ on the manifold using coordinates $\mathbf{u} \in \mathbb{R}^D$. The map T induces a metric $G(\mathbf{u})$ on the tangent space of $\mathcal{X}$ at $\mathbf{x} = T(\mathbf{u})$, given by (Kobayashi and Nomizu, 1963):

$$G(\mathbf{u}) = J_T(\mathbf{u})^\top J_T(\mathbf{u}). \tag{104}$$

As a result, an infinitesimal volume on $\mathcal{X}$ is given by $d\nu(\mathbf{x}) = \sqrt{\det G(\mathbf{u})}\, d\mathbf{u}$. A formula relating the density on $\mathcal{X}$ to that on the Euclidean space $\mathbb{R}^D$ can be derived from the conservation of measure (Equation 88) by setting $\mathcal{U} = \mathbb{R}^D$, taking $d\mu(\mathbf{u})$ to be the Lebesgue measure on $\mathbb{R}^D$, and reparameterizing $d\nu(\mathbf{x}) = \sqrt{\det G(\mathbf{u})}\, d\mathbf{u}$, which yields:

$$\int_\omega p_{\mathrm{u}}(\mathbf{u})\, d\mathbf{u} = \int_\omega p_{\mathrm{x}}(T(\mathbf{u}))\sqrt{\det G(\mathbf{u})}\, d\mathbf{u}. \tag{105}$$

Since the above must be true for any $\omega \subseteq \mathbb{R}^D$, it follows that:

$$p_{\mathrm{u}}(\mathbf{u}) = p_{\mathrm{x}}(T(\mathbf{u}))\sqrt{\det G(\mathbf{u})}, \tag{106}$$

which gives the density on $\mathbb{R}^D$ as a function of the density on the manifold. If we restrict the range of T to $\mathcal{X}$, we can define the inverse mapping $T^{-1} : \mathcal{X} \to \mathbb{R}^D$ and then use it to obtain the density on the manifold:

$$p_{\mathrm{x}}(\mathbf{x}) = p_{\mathrm{u}}\big(T^{-1}(\mathbf{x})\big)\left[\det G\big(T^{-1}(\mathbf{x})\big)\right]^{-1/2}. \tag{107}$$

The usual density-transformation formula for flows on $\mathbb{R}^D$ is a particular case of Equation 107. Taking $\mathcal{X} = \mathbb{R}^D$ and $M = D$, the Jacobian $J_T(\mathbf{u})$ becomes $D \times D$, and the infinitesimal volume on $\mathcal{X}$ simplifies to:

$$d\nu(\mathbf{x}) = \sqrt{(\det J_T(\mathbf{u}))^2}\, d\mathbf{u} = |\det J_T(\mathbf{u})|\, d\mathbf{u}, \tag{108}$$

which retrieves the standard flow on $\mathbb{R}^D$.

Using the above approach, we can define flows for which both $\mathcal{U}$ and $\mathcal{X}$ are D-dimensional Riemannian manifolds. We start from a base density defined on a manifold $\mathcal{U}$, transform

it to $\mathbb{R}^D$ using the inverse embedding map for $\mathcal{U}$, perform any number of standard flow steps on $\mathbb{R}^D$, and finally transform the resulting density on the target manifold $\mathcal{X}$ using the embedding map for $\mathcal{X}$. We illustrate this approach in Figure 9, where $\mathcal{U}$ and $\mathcal{X}$ are 2-dimensional spheres embedded in $\mathbb{R}^3$.

An important limitation of the above approach is that it assumes the existence of a differentiable invertible map $T : \mathbb{R}^D \to \mathcal{X}$, whose inverse maps the manifold onto $\mathbb{R}^D$. However, such a map can only exist for manifolds that are homeomorphic, i.e. topologically equivalent, to $\mathbb{R}^D$ (Kobayashi and Nomizu, 1963). There are several manifolds that are not homeomorphic to $\mathbb{R}^D$, including spheres, tori, and arbitrary products thereof, for which the above approach will be problematic. In the sphere example in Figure 9, any embedding map will create at least one coordinate singularity (akin to how all longitude lines meet at Earth's poles). Such coordinate singularities can manifest as points of infinite density, and thus create numerical instabilities in practice.

An alternative approach was proposed by Falorsi et al. (2019) for the special case where the manifold is also a group, in which case it is known as a *Lie group*. A Lie group can be reparameterized with respect to its Lie algebra (i.e. the tangent space at the identity element) using the exponential map, which maps the Lie algebra to the Lie group. Falorsi et al. (2019) show that this parameterization can be done analytically in certain cases; in general however, computing the exponential map has a cubic cost with respect to the dimensionality D.

5.5 Bypassing Topological Constraints

Due to T being a diffeomorphism, a flow must preserve topological properties, which means that $\mathcal{U}$ and $\mathcal{X}$ must be homeomorphic, i.e. topologically equivalent (Kobayashi and Nomizu, 1963). For example, a flow cannot map $\mathbb{R}^D$ to the sphere $\mathbb{S}^D$, or $\mathbb{R}^D$ to $\mathbb{R}^{D'}$ for $D \neq D'$, as these spaces have different topologies. For the same reason, every intermediate space $\mathcal{Z}_k$ along the flow's trajectory must be topologically equivalent to both $\mathcal{U}$ and $\mathcal{X}$. In the previous section, we discussed how this constraint can create problems for flows that map between Riemannian manifolds and Euclidean space.

This constraint's effect on continuous-time flows is prominent. In this case there is a continuum of intermediate spaces $\mathcal{Z}_t$ for $t \in [t_0, t_1]$, all of which must be topologically equivalent. For example, a continuous-time flow cannot implement a transformation $T : \mathbb{R} \to \mathbb{R}$ with $T(1) = -1$ and $T(-1) = 1$, due to any such transformation requiring an intersection at some intermediate stage (Dupont et al., 2019). One way to bypass this restriction is to instantiate the flow in a lifted space, as was proposed by Dupont et al. (2019). Introducing ρ auxiliary variables, the flow's transformation is now a map between $D + \rho$ dimensional spaces, $T : \mathbb{R}^{D+\rho} \to \mathbb{R}^{D+\rho}$. Although the lifted spaces must still be topologically equivalent, T can now represent a wider class of functions when projected to the original D-dimensional space. However, density evaluation in the original D-dimensional space can no longer be done analytically, which removes a benefit of flows that makes them an attractive modeling choice in the first place. Rather, the ρ auxiliary variables must be numerically integrated in practice. Dupont et al. (2019)'s augmentation method can be thought of as defining a mix-

ture of flows whereby the auxiliary variables serve as an index; integrating out the auxiliary variables is then akin to summing over the (infinitely many) mixture components.

For another example, if the target density is comprised of disconnected modes, the base density must have the same number of disconnected modes. This is true for all flows, not just continuous-time ones. If the base density has fewer modes than the target (which will likely be the case in practice), the flow will be forced to assign a non-zero amount of probability mass to the 'empty' space between the disconnected modes. To alleviate this issue, Cornish et al. (2019) propose a latent-variable flow defined as follows:

$$\mathbf{u} \sim p_{\mathrm{u}}(\mathbf{u}), \quad \mathbf{z} \sim p(\mathbf{z} \,|\, \mathbf{u}), \quad \mathbf{x} = T(\mathbf{u}; \mathbf{z}), \tag{109}$$

where $\mathbf{z}$ is a latent variable. Just as with the auxiliary-variable formulation of Dupont et al. (2019), we can think of the above as a mixture of flows, with $\mathbf{z}$ indexing each mixture component. Additionally, the above formulation is the same as RAD (Dinh et al., 2019) discussed in Section 5.2, except that here $\mathbf{z}$ can be continuous and the mixture components can overlap. However, $p_{\mathrm{x}}(\mathbf{x})$ can no longer be calculated analytically in general. Cornish et al. (2019) make do by performing variational inference, introducing an approximation $q(\mathbf{z} \,|\, \mathbf{x})$ that then allows for optimization via a lower bound on $\log p_{\mathrm{x}}(\mathbf{x})$.

5.6 Symmetric Densities and Equivariant Flows

In many real-world applications, we have domain knowledge of the symmetries of the target density. For example, if we want to model a physical system composed of many interacting particles, the physical interactions between particles (e.g. gravitational or electrostatic forces) are invariant to translations. Another example is modeling the configuration of a molecular structure, where there may be rotational symmetries in the molecule. Such symmetries result in a probability density (e.g. of molecular configurations) that is invariant to specific transformations (e.g. rotations).

In such situations, it is desirable to build the known symmetries directly into the model, not only so that the model has the right properties, but also so that estimating the model is more data efficient. However, in general it is not trivial to build expressive probability densities that are also invariant to a prescribed set of symmetries. Flows are a good candidate class of models to combine with such domain knowledge because we have a lot of control of how they deform an initial density. In this section, we explore ways to build flow-based models that respect a prescribed set of symmetries.

We say that g is a *symmetry* of the density $p : \mathbb{R}^D \to [0, +\infty)$ with representation an invertible matrix $\mathbf{R}_g \in \mathbb{R}^{D \times D}$ if the density remains invariant after transforming its input by $\mathbf{R}_g$, that is, if $p(\mathbf{R}_g \mathbf{x}) = p(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{R}^D$. For example, a standard-normal density is symmetric with respect to rotations, reflections, and axes permutations. It is easy to show that $|\det \mathbf{R}_g| = 1$ for any symmetry g. Using the change of variables $\mathbf{x}' = \mathbf{R}_g^{-1}\mathbf{x}$, we have the following:

$$1 = \int p(\mathbf{x})\, d\mathbf{x} = \int p(\mathbf{R}_g \mathbf{x}')\, |\det \mathbf{R}_g|\, d\mathbf{x}' = |\det \mathbf{R}_g| \int p(\mathbf{x}')\, d\mathbf{x}' = |\det \mathbf{R}_g|\,. \tag{110}$$

The set of all symmetries of a target density is closed under composition, is associative, has an identity element, and each element has an inverse—therefore forming a group G.

An important concept for dealing with symmetries in group theory is that of *equivariance*. We say that a transformation $T : \mathbb{R}^D \to \mathbb{R}^D$ is equivariant with respect to the group G if $T(\mathbf{R}_g\mathbf{u}) = \mathbf{R}_g T(\mathbf{u})$ for all $g \in G$ and $\mathbf{u} \in \mathbb{R}^D$—that is, we get the same result regardless of whether we transform the input or the output of T by $\mathbf{R}_g$. It is straightforward to see that the composition of two equivariant transformations is also equivariant, and if T is bijective its inverse is also equivariant.

These observations lead to the result shown in Lemma 1, which provides a general mechanism for constructing flow-based models whose density is invariant with respect to a prescribed symmetry group G.

Lemma 1 (Equivariant flows) *Let $p_{\mathrm{x}}(\mathbf{x})$ be the density function of a flow-based model with transformation $T : \mathbb{R}^D \to \mathbb{R}^D$ and base density $p_{\mathrm{u}}(\mathbf{u})$. If T is equivariant with respect to G and $p_{\mathrm{u}}(\mathbf{u})$ is invariant with respect to G, then $p_{\mathrm{x}}(\mathbf{x})$ is invariant with respect to G.*

Proof The flow density evaluated at $\mathbf{R}_g\mathbf{x}$ for some $g \in G$ is equal to:

$$p_{\mathrm{x}}(\mathbf{R}_g\mathbf{x}) = p_{\mathrm{u}}\big(T^{-1}(\mathbf{R}_g\mathbf{x})\big)\, \left|\det J_{T^{-1}}(\mathbf{R}_g\mathbf{x})\right|. \tag{111}$$

From the equivariance of T and hence of T^{-1}, we have that $T^{-1}(\mathbf{R}_g\mathbf{x}) = \mathbf{R}_g T^{-1}(\mathbf{x})$. Taking the Jacobian of both sides, we obtain:

$$\begin{aligned} J_{T^{-1}}(\mathbf{R}_g\mathbf{x})\,\mathbf{R}_g = \mathbf{R}_g J_{T^{-1}}(\mathbf{x}) \;&\Rightarrow\; \left|\det J_{T^{-1}}(\mathbf{R}_g\mathbf{x})\right| \left|\det \mathbf{R}_g\right| = \left|\det \mathbf{R}_g\right| \left|\det J_{T^{-1}}(\mathbf{x})\right| \\ &\Rightarrow\; \left|\det J_{T^{-1}}(\mathbf{R}_g\mathbf{x})\right| = \left|\det J_{T^{-1}}(\mathbf{x})\right|. \end{aligned}$$

Finally, from the invariance of $p_{\mathrm{u}}(\mathbf{u})$ we have that $p_{\mathrm{u}}\big(\mathbf{R}_g T^{-1}(\mathbf{x})\big) = p_{\mathrm{u}}\big(T^{-1}(\mathbf{x})\big)$. Therefore, $p_{\mathrm{x}}(\mathbf{R}_g\mathbf{x}) = p_{\mathrm{x}}(\mathbf{x})$ for all $g \in G$ and $\mathbf{x} \in \mathbb{R}^D$. ■

The concept of equivariant flows was introduced concurrently by Köhler et al. (2019) and Rezende et al. (2019). Both of these works explored equivariance in the context of continuous-time flows. Köhler et al. (2019) provided a concrete example of a continuous-time flow that is equivariant with respect to rigid translations and rotations, whereas Rezende et al. (2019) proposed a general method for enforcing equivariance using known symmetry generators.

In practice, taking the base density to be invariant with respect to a prescribed symmetry group is usually easy. What it more challenging is constructing the transformation T to be equivariant. One approach is based on invariant functions with respect to G, that is, functions f for which $f(\mathbf{R}_g\mathbf{u}) = f(\mathbf{u})$ for all $g \in G$ and $\mathbf{u} \in \mathbb{R}^D$. This approach is outlined in Lemma 2 below.

Lemma 2 (Equivariance from invariance) *Let $f : \mathbb{R}^D \to \mathbb{R}$ be invariant with respect to G, and assume that $\mathbf{R}_g$ is orthogonal for all $g \in G$. Then $\nabla_{\mathbf{u}} f(\mathbf{u})$ is equivariant with respect to G.*

Proof From the invariance of f we have that $f(\mathbf{R}_g\mathbf{u}) = f(\mathbf{u})$. Taking the gradient on both sides, we obtain:

$$\begin{aligned}\mathbf{R}_g^\top \nabla_{\mathbf{R}_g\mathbf{u}} f(\mathbf{R}_g\mathbf{u}) = \nabla_{\mathbf{u}} f(\mathbf{u}) \quad &\Rightarrow \quad \nabla_{\mathbf{R}_g\mathbf{u}} f(\mathbf{R}_g\mathbf{u}) = \mathbf{R}_g^{-\top} \nabla_{\mathbf{u}} f(\mathbf{u}) \\ &\Rightarrow \quad \nabla_{\mathbf{R}_g\mathbf{u}} f(\mathbf{R}_g\mathbf{u}) = \mathbf{R}_g \nabla_{\mathbf{u}} f(\mathbf{u}).\end{aligned}$$

■

Lemma 2 gives a general mechanism for constructing equivariant transformations with respect to symmetries with orthogonal representations. Several symmetries fall into this category, including rotations, reflections and axes permutations. The practical significance of Lemma 2 is that it is often easier to construct an invariant function than an equivariant one. For instance, any function of the ℓ_2 norm $\|\mathbf{u}\|$ is invariant with respect to symmetries with orthogonal representations, since $\|\mathbf{R}\mathbf{u}\| = \|\mathbf{u}\|$ for any orthogonal matrix $\mathbf{R}$.

6. Applications

Normalizing flows have two primitive operations: density calculation and sampling. In turn, flows are effective in any application requiring a probabilistic model with either of those capabilities. In this section, we summarize applications to probabilistic modeling, inference, supervised learning, and reinforcement learning.

6.1 Probabilistic Modeling

Normalizing flows, due to their ability to be expressive while still allowing for exact likelihood calculations, are often used for probabilistic modeling of data. For this application, we assume access to a finite number of draws $\mathbf{x}$ from some unknown generative process $p^*_{\mathrm{x}}(\mathbf{x})$. These draws constitute a size N data set $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^N$. Our goal then is to fit a flow-based model $p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})$ to $\mathbf{X}$ such that the model serves as a good approximation for $p^*_{\mathrm{x}}(\mathbf{x})$.

Often, the data $\{\mathbf{x}_n\}_{n=1}^N$ are discrete; for example, they could be images with pixel values in $\{0, 1, \ldots, 255\}$. Flow-based models are defined over continuous random variables (with the exception of discrete flows, Section 5.3), so they are not directly applicable to discrete data. To use flows with discrete data, we often dequantize $\{\mathbf{x}_n\}_{n=1}^N$ by adding continuous noise. The noise distribution can be fixed (e.g. uniform in $[0, 1]$ for the image example above), or learned, as for example in *variational dequantization* (Ho et al., 2019).

One of the most popular method for fitting $p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})$ is *maximum likelihood estimation*, which exploits the forward KL divergence first introduced in Section 2.3.1:

$$\begin{aligned}D_{\mathrm{KL}}\left[\, p^*_{\mathrm{x}}(\mathbf{x}) \,\|\, p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta}) \,\right] &= -\mathbb{E}_{p^*_{\mathrm{x}}(\mathbf{x})}\left[\, \log p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta}) \,\right] + \text{const} \\ &\approx -\frac{1}{N}\sum_{n=1}^{N} \log p_{\mathrm{x}}(\mathbf{x}_n;\boldsymbol{\theta}) + \text{const} \\ &= -\frac{1}{N}\sum_{n=1}^{N} \log p_{\mathrm{u}}(T^{-1}(\mathbf{x}_n;\boldsymbol{\phi});\boldsymbol{\psi}) + \log\left|J_{T^{-1}}(\mathbf{x}_n;\boldsymbol{\phi})\right| + \text{const}.\end{aligned} \tag{112}$$

However, in principle any valid divergence or integral probability metric can be used as an optimization target, as discussed in Section 2.3.4. Typically, there are two downstream uses for the resulting model $p_{\mathrm{x}}(\mathbf{x};\boldsymbol{\theta})$: *density estimation* and *generation*.

6.1.1 Density Estimation

The first task is primarily quantitative: we can use the model to estimate densities, expectations, marginals, or other quantities of interest on never-before-seen data. Early work (Chen and Gopinath, 2000; Tabak and Turner, 2013) considered only synthetic low-dimensional cases, showing that normalizing flows could indeed represent skewed, multi-modal densities as well as kernel density estimators could. It was Laparra et al. (2011) that first applied Gaussianization to real data, using the density function to perform one-class classification to detect urban areas in satellite images. Rippel and Adams (2013) next showed that their deep flow model's density could detect rotations and corruptions of images. Papamakarios et al. (2017) performed a systematic comparison of their masked autoregressive flow on unconditional and conditional density estimation tasks, showing that the composition enabled by their framework allows for better density estimation than other variants (namely MADE and Real NVP). Grathwohl et al. (2019) performed similar experiments to validate the effectiveness of continuous-time flows.

6.1.2 Generation

The second task is generation: sampling from the model novel instances that could have plausibly been sampled from $p^*_{\mathrm{x}}(\mathbf{x})$. In this latter case, the availability of exact likelihood values is not the end goal so much as a principled training target that one would expect to result in good generative performance. Generation has been a popular application of flows in machine learning, and below we summarize their use for various categories of data.

Images & video Image generation has been given serious effort since the earliest work on flows. Laparra et al. (2011), in the same work mentioned above, used Gaussianization to generate gray-scale images of faces. Rippel and Adams (2013) also demonstrated early success in generative performance as MNIST samples from their model looked rather compelling. Dinh et al. (2015), through the use of their coupling parameterization, showed further improvements including density estimation competitive with other high-capacity models (such as deep mixtures of factor analysers) and respectable generation of SVHN digits. In follow-up work, Dinh et al. (2017) increased the capacity of their model by including scale transformations (instead of just translations), being the first to demonstrate that flows could produce sharp, visually compelling full-color images. Specifically, Dinh et al. (2017) showed compelling samples from models trained on CelebA, ImageNet (64×64), CIFAR-10, and LSUN. Kingma and Dhariwal (2018), using a similar model but with additional convolutional layers, further improved upon Dinh et al. (2017)'s results in density estimation and generation of high-dimensional images. Continuous (Grathwohl et al., 2019) and residual (Behrmann et al., 2019; Chen et al., 2019) flows have been demonstrated to produce sharp, high-dimensional images as well. Finally, Kumar et al. (2019) propose a normalizing flow

for modeling video data, adapting the Glow architecture (Kingma and Dhariwal, 2018) to synthesize raw RGB frames.

Audio The autoregressive model *WaveNet* (van den Oord et al., 2016a) demonstrated impressive performance in audio synthesis. While WaveNet is not a normalizing flow, in follow-up work van den Oord et al. (2018) defined a proper flow for audio synthesis by distilling WaveNet into an inverse autoregressive flow so as to make test-time sampling more efficient. Prenger et al. (2019) and Kim et al. (2019) have since formulated WaveNet variants built from coupling layers to enable fast likelihood and sampling, in turn obviating the need for van den Oord et al. (2018)'s post-training distillation step.

Text The most direct way to apply normalizing flows to text data is to define a discrete flow over characters or a vocabulary. Tran et al. (2019) take this approach, showing performance in character-level language modeling competitive to RNNs while having superior generation runtime. An alternative approach that has found wider use is to define a latent variable model with a discrete likelihood but a continuous latent space. A normalizing flow can then be defined on the latent space as usual. Ziegler and Rush (2019) use such an approach for character-level language modeling. Zhou et al. (2019), He et al. (2018), and Jin et al. (2019) define normalizing flows on the continuous space of word embeddings as a subcomponent of models for translation, syntactic structure, and parsing respectively.

Other structured objects Extending flows to operate on other structured objects is a burgeoning area of work. So far, flows have been applied to graphs (Deng et al., 2019), molecules (Madhawa et al., 2019; Honda et al., 2019), point clouds (Yang et al., 2019), and part models for motion synthesis (Henter et al., 2019).

6.2 Inference

In the previous section our focus was on modeling data and recovering its underlying distribution. We now turn to inference: estimating unknown quantities within a model. The most common setting is the computation of high-dimensional, analytically intractable integrals of the form:

$$\int \pi(\boldsymbol{\eta})\, d\boldsymbol{\eta}. \tag{113}$$

Bayesian inference usually runs into such an obstacle when computing the posterior's normalizing constant or when computing expectations under the posterior. Below we summarize the use of flows for sampling, variational inference, and likelihood-free inference.

6.2.1 Importance and Rejection Sampling

Importance sampling (IS) computes intractable integrals by converting them to an expectation under an auxiliary distribution $q(\boldsymbol{\eta})$:

$$\int \pi(\boldsymbol{\eta})\, d\boldsymbol{\eta} = \int q(\boldsymbol{\eta}) \frac{\pi(\boldsymbol{\eta})}{q(\boldsymbol{\eta})}\, d\boldsymbol{\eta} = \mathbb{E}_{q(\boldsymbol{\eta})}\left[\frac{\pi(\boldsymbol{\eta})}{q(\boldsymbol{\eta})}\right] \approx \frac{1}{S}\sum_{s=1}^{S} \frac{\pi(\hat{\boldsymbol{\eta}}_s)}{q(\hat{\boldsymbol{\eta}}_s)}, \tag{114}$$

where $q(\boldsymbol{\eta})$ is a user-specified density function and $\hat{\boldsymbol{\eta}}_s$ is a sample from $q(\boldsymbol{\eta})$. Clearly, IS requires both sampling and density evaluation. Since both operations are tractable for many flows, they make for an attractive model from which to construct the proposal.

Müller et al. (2019) do just this: they implement $q(\boldsymbol{\eta})$ using normalizing flows. The practicality of IS crucially depends on the choice of proposal and thus the flow's parameters must be optimized. Müller et al. (2019) discuss two strategies for fitting $q(\boldsymbol{\eta})$. When $\pi(\boldsymbol{\eta})$ can be interpreted as an unnormalized density, the first is to minimize the KL divergence between the normalized target and the flow: $D_{\mathrm{KL}}\left[\, p(\boldsymbol{\eta}) \,\|\, q(\boldsymbol{\eta}) \,\right]$ where $p(\boldsymbol{\eta}) = \pi(\boldsymbol{\eta})/Z$ with $Z = \int \pi(\boldsymbol{\eta})\, d\boldsymbol{\eta}$ being an intractable normalizing constant (equal to the very quantity that we wish to compute). While $D_{\mathrm{KL}}\left[\, p(\boldsymbol{\eta}) \,\|\, q(\boldsymbol{\eta}) \,\right]$ cannot be computed, IS can be used to compute the divergence's gradient with respect to the flow's parameters. The second is to minimize the variance of the IS estimator directly. When $\pi(\boldsymbol{\eta})$ is again an unnormalized density, this is equivalent to minimizing a χ^2-divergence between the proposal and $p(\boldsymbol{\eta}) = \pi(\boldsymbol{\eta})/Z$. Flows have also been used for the proposal distribution in similar ways by Noé et al. (2019) and Wirnsberger et al. (2020).

The related technique of *rejection sampling* (RS) aims to draw samples from $p(\boldsymbol{\eta}) = \pi(\boldsymbol{\eta})/Z$, where $\pi(\boldsymbol{\eta})$ is again an unnormalized density. Both density evaluation and sampling are required from the proposal in RS, again making normalizing flows well-suited. Bauer and Mnih (2019) use the Real NVP architecture (Dinh et al., 2017) to parameterize a proposal distribution for RS since the coupling layers allow for fast density evaluation and sampling.

6.2.2 Markov Chain Monte Carlo

The application of flows in *Markov chain Monte Carlo* (MCMC) precedes the appearance of flows in deep learning by at least a few decades. One prominent example is *Hamiltonian Monte Carlo* (HMC), also known as *Hybrid Monte Carlo* (Duane et al., 1987; Neal, 2010). HMC operates on the 'phase space' $(\boldsymbol{\eta}, \mathbf{v})$, where $\boldsymbol{\eta}$ are the variables of interest and $\mathbf{v}$ are additional 'momentum' variables of the same dimensionality as $\boldsymbol{\eta}$. HMC generates samples from a joint distribution $p(\boldsymbol{\eta}, \mathbf{v})$ constructed so that its marginal over $\boldsymbol{\eta}$ is the distribution of interest. Central to HMC is the *Hamiltonian* defined by $H(\boldsymbol{\eta}, \mathbf{v}) = -\log p(\boldsymbol{\eta}, \mathbf{v})$. Given a state $(\boldsymbol{\eta}, \mathbf{v})$, HMC proposes a new state $(\boldsymbol{\eta}', \mathbf{v}') = T(\boldsymbol{\eta}, \mathbf{v})$ deterministically, where T is a *Hamiltonian flow* followed by negation of the momentum variables. The proposed state is then accepted/rejected using the usual Metropolis–Hastings step. The Hamiltonian flow is the continuous-time flow generated by the following ODE:

$$\frac{d(\boldsymbol{\eta}, \mathbf{v})}{dt} = \left(\frac{\partial H}{\partial \mathbf{v}}, -\frac{\partial H}{\partial \boldsymbol{\eta}} \right). \tag{115}$$

This flow is volume-preserving, meaning that its absolute Jacobian determinant is 1 everywhere, which, in combination with the negation of the momentum variables, ensures that the proposal is symmetric and thus cancels in the Metropolis–Hastings ratio.

It is also possible to construct MCMC algorithms with flows other than the Hamiltonian flow described above. One example is A-NICE-MC (Song et al., 2017), which is similar to HMC

but constructs the proposal using an arbitrary volume-preserving flow $T(\cdot;\boldsymbol{\phi})$ parameterized by $\boldsymbol{\phi}$. Song et al. (2017) use the NICE model of Dinh et al. (2015), but their method applies to any volume-preserving flow more generally. Given a state $(\boldsymbol{\eta}, \mathbf{v})$, a new state is proposed which is equal to either $T(\boldsymbol{\eta}, \mathbf{v}; \boldsymbol{\phi})$ or $T^{-1}(\boldsymbol{\eta}, \mathbf{v}; \boldsymbol{\phi})$ with equal probability. This proposal is symmetric and so it cancels in the Metripolis–Hastings ratio. The parameters $\boldsymbol{\phi}$ are tuned to the distribution of interest using adversarial training.

Another way of applying flows to MCMC is to use the flow to reparameterize the target distribution. It is well understood that the efficiency of MCMC drastically depends on the target distribution being easy to explore. If the target is highly skewed and/or multi-modal, the performance of MCMC suffers, resulting in slow mixing and convergence. Normalizing flows can effectively 'smooth away' these pathologies in the target's geometry by allowing MCMC to be run on the simpler and better-behaved base density. Given the unnormalized target $\pi(\boldsymbol{\eta})$, we can reparameterize the model in terms of a base density $p_{\mathrm{u}}(\mathbf{u})$ such that $\boldsymbol{\eta} = T(\mathbf{u}; \boldsymbol{\phi})$. Assuming a symmetric proposal distribution for simplicity, applying the Metropolis–Hastings ratio to the reparameterized model yields:

$$r(\hat{\mathbf{u}}_*; \hat{\mathbf{u}}_t) = \frac{p_{\mathrm{u}}(\hat{\mathbf{u}}_*)}{p_{\mathrm{u}}(\hat{\mathbf{u}}_t)} = \frac{\pi(T(\hat{\mathbf{u}}_*; \boldsymbol{\phi}))\ |\det J_T(\hat{\mathbf{u}}_*; \boldsymbol{\phi})|}{\pi(T(\hat{\mathbf{u}}_t; \boldsymbol{\phi}))\ |\det J_T(\hat{\mathbf{u}}_t; \boldsymbol{\phi})|}, \tag{116}$$

where $\hat{\mathbf{u}}_*$ denotes the proposed value and $\hat{\mathbf{u}}_t$ the current value. Assuming T is sufficiently powerful such that $T^{-1}(\boldsymbol{\eta}; \boldsymbol{\phi})$ is truly distributed according to the simpler base distribution, exploring the target should become considerably easier. In practice, it is still useful to generate proposals via Hamiltonian dynamics rather than from a simple isotropic proposal, even if $p_{\mathrm{u}}(\mathbf{u})$ is isotropic (Hoffman et al., 2019).

While the reparameterization above is relatively straightforward, there is still the crucial issue of how to set or optimize the parameters of T. Titsias (2017) interleaves runs of the chain with updates to the flow's parameters, performing optimization by maximizing the unnormalized reparameterized target under the last sample from a given run:

$$\arg\max_{\boldsymbol{\phi}} \log \pi(T(\hat{\mathbf{u}}_{t_{\text{final}}}; \boldsymbol{\phi})) + \log |\det J_T(\hat{\mathbf{u}}_{t_{\text{final}}}; \boldsymbol{\phi})| \,. \tag{117}$$

However, this choice is a heuristic, and is not guaranteed to encourage the chain's mixing. As Hoffman et al. (2019) point out, such a choice may emphasize mode finding. As an alternative, Hoffman et al. (2019) propose fitting the flow model to $p(\boldsymbol{\eta}) = \pi(\boldsymbol{\eta})/Z$ first via variational inference and then running Hamiltonian Monte Carlo on the reparameterized model, using a sample from the flow to initialize the chain.

6.2.3 Variational Inference

We can also use normalizing flows to fit distributions over latent variables or model parameters. Specifically, flows can usefully serve as posterior approximations for local (Rezende and Mohamed, 2015; van den Berg et al., 2018; Kingma et al., 2016; Tomczak and Welling, 2016) and global (Louizos and Welling, 2017) variables.

For example, suppose we wish to infer variables $\boldsymbol{\eta}$ given some observation $\mathbf{x}$. In variational inference with normalizing flows, we use a (trained) flow-based model $q(\boldsymbol{\eta}; \boldsymbol{\phi})$ to approximate

the posterior as follows:

$$p(\boldsymbol{\eta} \,|\, \mathbf{x}) \approx q(\boldsymbol{\eta}; \boldsymbol{\phi}) = q_{\mathrm{u}}(\mathbf{u}) \; |\det J_T(\mathbf{u}; \boldsymbol{\phi})|^{-1} \,, \tag{118}$$

where $q_{\mathrm{u}}(\mathbf{u})$ is the base distribution (which here is typically fixed) and $T(\cdot; \boldsymbol{\phi})$ is the transformation (parameterized by $\boldsymbol{\phi}$). If we want to approximate the posterior for multiple values of $\mathbf{x}$, we can make the flow model conditional on $\mathbf{x}$ and amortize the cost of inference across values of $\mathbf{x}$. The flow is trained by maximizing the *evidence lower bound* (ELBO), which can be written as:

$$\begin{aligned} \log p(\mathbf{x}) &\geq \mathbb{E}_{q(\boldsymbol{\eta};\boldsymbol{\phi})} \left[\log p(\mathbf{x}, \boldsymbol{\eta}) \right] - \mathbb{E}_{q(\boldsymbol{\eta};\boldsymbol{\phi})} \left[\log q(\boldsymbol{\eta}; \boldsymbol{\phi}) \right] \\ &= \mathbb{E}_{q_{\mathrm{u}}(\mathbf{u})} \left[\log p(\mathbf{x}, T(\mathbf{u}; \boldsymbol{\phi})) \right] - \mathbb{E}_{q_{\mathrm{u}}(\mathbf{u})} \left[\log q_{\mathrm{u}}(\mathbf{u}) \right] + \mathbb{E}_{q_{\mathrm{u}}(\mathbf{u})} \left[\log |\det J_T(\mathbf{u}; \boldsymbol{\phi})| \right] \\ &= \mathbb{E}_{q_{\mathrm{u}}(\mathbf{u})} \left[\log p(\mathbf{x}, T(\mathbf{u}; \boldsymbol{\phi})) \right] + \mathbb{H} \left[q_{\mathrm{u}}(\mathbf{u}) \right] + \mathbb{E}_{q_{\mathrm{u}}(\mathbf{u})} \left[\log |\det J_T(\mathbf{u}; \boldsymbol{\phi})| \right], \end{aligned} \tag{119}$$

where $\mathbb{H}\left[q_{\mathrm{u}}(\mathbf{u}) \right]$ is the differential entropy of the base distribution, which is a constant with respect to $\boldsymbol{\phi}$. The expectation terms can be estimated by Monte Carlo, using samples from the base distribution as follows:

$$\begin{aligned} \mathbb{E}_{q_{\mathrm{u}}(\mathbf{u})} \left[\log p(\mathbf{x}, T(\mathbf{u}; \boldsymbol{\phi})) \right] &\approx \frac{1}{S} \sum_{s=1}^{S} \log p(\mathbf{x}, T(\hat{\boldsymbol{u}}_s; \boldsymbol{\phi})), \\ \mathbb{E}_{q_{\mathrm{u}}(\mathbf{u})} \left[\log |\det J_T(\mathbf{u}; \boldsymbol{\phi})| \right] &\approx \frac{1}{S} \sum_{s=1}^{S} \log |\det J_T(\hat{\boldsymbol{u}}_s; \boldsymbol{\phi})| \,. \end{aligned} \tag{120}$$

Normalizing flows can be thought of as implementing a 'generalized reparameterization trick' (Kingma and Welling, 2014a; Rezende et al., 2014; Kingma and Welling, 2014b), as they leverage a transformation of a fixed distribution to draw samples from a distribution of interest. Flows therefore define flexible approximate posteriors that are readily reparameterizable by design.

6.2.4 Likelihood-Free Inference

Models are often implicit, meaning they are not defined in terms of a likelihood function $p(\mathbf{x} \,|\, \boldsymbol{\eta})$ that describes how observable variables $\mathbf{x}$ depend on model parameters $\boldsymbol{\eta}$. Rather, they come in the form of a simulator that takes in parameters $\boldsymbol{\eta}$ and simulates variables $\mathbf{x}$ (Diggle and Gratton, 1984). Such simulator-based models are common in scientific fields such as cosmology (Alsing et al., 2018), high-energy physics (Brehmer et al., 2018), and computational neuroscience (Gonçalves et al., 2020). Inferring the parameters $\boldsymbol{\eta}$ of a simulator-based model given observed data $\mathbf{x}$ is often referred to as *likelihood-free inference* (Papamakarios, 2019), *simulation-based inference* (Cranmer et al., 2020), or *approximate Bayesian computation* (Beaumont et al., 2002; Beaumont, 2010). The typical assumption in likelihood-free inference is that it is easy to simulate variables $\mathbf{x}$ from the model given $\boldsymbol{\eta}$, but it is intractable to evaluate the likelihood $p(\mathbf{x} \,|\, \boldsymbol{\eta})$.

Normalizing flows are a natural fit for likelihood-free inference, especially flows conditioned on side information (e.g. Winkler et al., 2019; Ardizzone et al., 2019). Assuming

a tractable prior distribution $p(\boldsymbol{\eta})$ over the parameters of interest, we can generate a data set $\{(\boldsymbol{\eta}_n, \mathbf{x}_n)\}_{n=1}^N$ where $\boldsymbol{\eta}_n \sim p(\boldsymbol{\eta})$ and $\mathbf{x}_n$ is simulated from the model with parameters $\boldsymbol{\eta}_n$. In other words, $(\boldsymbol{\eta}_n, \mathbf{x}_n)$ is a joint sample from $p(\boldsymbol{\eta}, \mathbf{x}) = p(\boldsymbol{\eta})\, p(\mathbf{x} \,|\, \boldsymbol{\eta})$. Then, using the techniques described in Section 6.1, we can fit a flow-based model $q(\boldsymbol{\eta} \,|\, \mathbf{x})$ conditioned on $\mathbf{x}$ to the generated data set $\{(\boldsymbol{\eta}_n, \mathbf{x}_n)\}_{n=1}^N$ in order to approximate the posterior $p(\boldsymbol{\eta} \,|\, \mathbf{x})$ (Greenberg et al., 2019; Gonçalves et al., 2020). Alternatively, we can fit a flow-based model $q(\mathbf{x} \,|\, \boldsymbol{\eta})$ conditioned on $\boldsymbol{\eta}$ in order to approximate the intractable likelihood $p(\mathbf{x} \,|\, \boldsymbol{\eta})$ (Papamakarios et al., 2019). In either case, the trained flow-based model is useful for various downstream tasks that require density evaluation or sampling.

6.3 Using Flows for Representation Learning

Flows also have applications as building blocks for downstream tasks. We discuss two such cases, namely supervised learning and reinforcement learning.

6.3.1 Classification and Hybrid Modeling

Invertible ResNets—in addition to implementing residual flows, as discussed in Section 3.3—have been explored for classification (Jacobsen et al., 2018; Behrmann et al., 2019). This line of work exploits invertibility for other end-purposes than density computation. The first is for engineering improvements: to reduce the model's memory footprint by obviating the need to store activations for backpropagation (Gomez et al., 2017). The second is for improved model interpretability and understanding of the mechanics of deep learning. Jacobsen et al. (2018) showed that an invertible ResNet could be trained to nearly the same ImageNet accuracy as a non-invertible ResNet. This achievement may help us understand to what degree discarding information is crucial to deep learning's success (Tishby and Zaslavsky, 2015). Jacobsen et al. (2019) used invertible architectures to study the relationship between invariance and vulnerability to adversarial attacks.

Flows can be used for joint generative and predictive modeling by using them as the core component of a *hybrid model* (Nalisnick et al., 2019). Like in invertible ResNets, the flow is used as a deep, neural feature extractor, but unlike in ResNets, the architecture is chosen such that the Jacobian determinant is tractable. Leveraging the generative model $\mathbf{x} = T(\mathbf{u})$ to reparameterize the joint density, we can write:

$$\begin{aligned} \log p(\mathbf{y}, \mathbf{x}) &= \log p(\mathbf{y} \,|\, \mathbf{x}) + \log p(\mathbf{x}) \\ &= \log p\big(\mathbf{y} \,|\, T^{-1}(\mathbf{x})\big) + \log p_{\mathrm{u}}\big(T^{-1}(\mathbf{x})\big) + \log \left|\det J_{T^{-1}}(\mathbf{x})\right| . \end{aligned} \tag{121}$$

We can think of the flow as defining the first $L-1$ layers of the architecture, and of the last layer as a (generalized) linear model operating on the features $\mathbf{u} = T^{-1}(\mathbf{x})$. The second term then makes these features distribute according to $p_{\mathrm{u}}(\mathbf{u})$, which could be viewed as a regularizer on the feature space. For instance, if $p_{\mathrm{u}}(\mathbf{u})$ is standard Normal, then $\log p_{\mathrm{u}}\big(T^{-1}(\mathbf{x})\big)$ effectively acts as an ℓ_2 penalty. The Jacobian determinant serves its usual role in ensuring the density is properly normalized. Hence, a hybrid model in the form of Equation 121 can compute the joint density of labels and features at little additional cost to

standard forward propagation. The extra computation is introduced by the right-most two terms and depends on their particular forms. If $p_{\mathrm{u}}(\mathbf{u})$ is standard Normal and T defined via coupling layers, then the additional computation is $\mathcal{O}(DL)$ with D being the number of input dimensions and L the number of layers.

6.3.2 Reinforcement Learning

Finally, in this section we give two examples of how normalizing flows have been used thus far for reinforcement learning (RL).

Reparameterized policies The most popular use for flows in RL has been to model (continuous) policies. An action $\mathbf{a}_t \in \mathbb{R}^D$ taken in state $\mathbf{s}_t$ at time t is sampled according to $\hat{\mathbf{a}}_t = T(\hat{\mathbf{u}}_t; \mathbf{s}_t, \boldsymbol{\phi})$, $\hat{\mathbf{u}}_t \sim p_{\mathrm{u}}(\mathbf{u}_t)$ where p_{u} denotes the base density. The corresponding conditional density is written as:

$$pi(\mathbf{a}_t \,|\, \mathbf{s}_t) = p_{\mathrm{u}}(\mathbf{u}_t) \left|\det J_T(\mathbf{u}_t; \mathbf{s}_t, \boldsymbol{\phi})\right|^{-1}. \tag{122}$$

Haarnoja et al. (2018) and Ward et al. (2019) use such a policy within the maximum entropy and soft actor-critic frameworks respectively.

Imitation learning Schroecker et al. (2019) use normalizing flows within the imitation-learning paradigm for control problems. Given the observed expert (continuous) state-action pairs $(\bar{\mathbf{s}}, \bar{\mathbf{a}})$, a core challenge to imitation learning is accounting for unobserved intermediate states. Schroecker et al. (2019) use conditional flows to simulate these intermediate states and their corresponding actions. Specifically, for a state $\bar{\mathbf{s}}_{t+j}$ $(j \geq 1)$, predecessor state-action pairs are sampled from the model:

$$p(\mathbf{s}_t, \mathbf{a}_t \,|\, \bar{\mathbf{s}}_{t+j}) = p(\mathbf{a}_t \,|\, \mathbf{s}_t, \bar{\mathbf{s}}_{t+j})\, p(\mathbf{s}_t \,|\, \bar{\mathbf{s}}_{t+j}). \tag{123}$$

Both terms on the right-hand side are defined by MAFs (Papamakarios et al., 2017).

7. Conclusions

We have described normalizing flows and their use for probabilistic modeling and inference. We addressed key issues such as their expressive power (Section 2.2) and the fundamentals underlying their construction (both in discrete and continuous time). We also described the general principle of probability transformations (Section 5.1) and its implications for defining flows beyond Euclidean space. In particular, we showed that discrete domains, mixtures of flows, and extensions to Riemannian manifolds all follow from this generalized perspective. Lastly, we summarized the primary applications of flows (Section 6): tasks ranging from density estimation to likelihood-free inference to classification.

While many flow designs and specific implementations will inevitably become out-of-date as work on normalizing flows continues, we have attempted to isolate foundational ideas that will continue to guide the field well into the future. One of these keystone principles is the

chain rule of probability and its relationship to transformations with a triangular Jacobian. Autoregressive flows stand on these two pillars, with the former underlying their expressive power and the latter providing their efficient implementation. Similarly, the Banach fixed-point theorem provides the mathematical foundation for contractive residual flows. While alternative parameterizations of the translation function g_{ϕ} or normalization strategies may be developed, the underlying Lipschitz constraints cannot be deserted without violating the fixed-point theorem.

Throughout the text we emphasized crucial implementation notes that guide a successful application of flows. Perhaps of foremost importance is determining the computational constraints on evaluating the forward and inverse transformations. As we showed in Section 2, sampling and density evaluation place distinct demands on the transformation. Since flows have no inherent requirement as to whether T should implement $\mathcal{U} \to \mathcal{X}$ or vice versa, we are free to choose which direction better suits our application. If either sampling or density estimation is the primary objective—but not both—then autoregressive flows present an attractive, flexible class of model. Yet if both sampling and density evaluation must be done often or quickly, then implementing the autoregressive flow with a coupling-based conditioner will make both operations efficient at the cost of expressive power. On the other hand, non-autoregressive flows such as linear and residual flows allow for interaction between all dimensions at each step in the flow. While these interactions can be useful at times—making linear flows good for permuting variables between successive autoregressive flows and residual flows highly expressive—other limitations arise. For instance, contractive residual flows typically require iterative algorithms for sampling *and* density evaluation. As all flow constructions present trade-offs of some form, we hope this article provides a coherent and accessible summary to guide practitioners through these choice points.

Looking forward, the obstacles preventing wider application of normalizing flows are similar in spirit to those faced by any probabilistic model. However, unlike other probabilistic models that require approximate inference as they scale, flows usually admit analytical calculations and exact sampling even in high dimensions. Rather, the difficulty is transferred to the construction of the flow's transformation: how can we define ever more flexible transformations while keeping exact density evaluation and sampling computationally tractable? This is currently the focus of much work and will likely remain a core issue for some time. More study of the theoretical properties of flows is also needed. Understanding their approximation capabilities for finite sample and finite depth settings would help practitioners select which flow classes are best for a given application. Our discussion of generalizations in Section 5 will hopefully provide grounding as well as inspiration for this next wave of developments in the theory and application of normalizing flows.

Acknowledgments

We would like to thank Ivo Danihelka for his invaluable feedback on the manuscript, and the anonymous reviewers for their many improvement suggestions. We also thank Hyunjik Kim and Sébastien Racanière for useful discussions on a wide variety of flow-related topics.

Appendix A. Proof of KL Dualities

Let $p_{\mathrm{x}}(\mathbf{x})$ be the distribution induced by a flow with transformation T and base distribution $p_{\mathrm{u}}(\mathbf{u})$. Also, let $p^*_{\mathrm{u}}(\mathbf{u})$ be the distribution induced by the inverse flow with transformation T^{-1} and base distribution $p^*_{\mathrm{x}}(\mathbf{x})$. Using the formula for the density of a flow-based model and a change of variables, we have the following:

$$\begin{aligned} D_{\mathrm{KL}}\left[\, p^*_{\mathrm{x}}(\mathbf{x}) \,\|\, p_{\mathrm{x}}(\mathbf{x}) \,\right] &= \mathbb{E}_{p^*_{\mathrm{x}}(\mathbf{x})}\left[\, \log p^*_{\mathrm{x}}(\mathbf{x}) - \log p_{\mathrm{x}}(\mathbf{x}) \,\right] \\ &= \mathbb{E}_{p^*_{\mathrm{x}}(\mathbf{x})}\left[\, \log p^*_{\mathrm{x}}(\mathbf{x}) - \log \left|\det J_{T^{-1}}(\mathbf{x})\right| - \log p_{\mathrm{u}}\big(T^{-1}(\mathbf{x})\big) \,\right] \\ &= \mathbb{E}_{p^*_{\mathrm{u}}(\mathbf{u})}\left[\, \log p^*_{\mathrm{x}}(T(\mathbf{u})) + \log \left|\det J_{T}(\mathbf{u})\right| - \log p_{\mathrm{u}}(\mathbf{u}) \,\right] \\ &= \mathbb{E}_{p^*_{\mathrm{u}}(\mathbf{u})}\left[\, \log p^*_{\mathrm{u}}(\mathbf{u}) - \log p_{\mathrm{u}}(\mathbf{u}) \,\right] \\ &= D_{\mathrm{KL}}\left[\, p^*_{\mathrm{u}}(\mathbf{u}) \,\|\, p_{\mathrm{u}}(\mathbf{u}) \,\right]. \end{aligned} \tag{124}$$

Similarly, we have the following:

$$\begin{aligned} D_{\mathrm{KL}}\left[\, p_{\mathrm{x}}(\mathbf{x}) \,\|\, p^*_{\mathrm{x}}(\mathbf{x}) \,\right] &= \mathbb{E}_{p_{\mathrm{x}}(\mathbf{x})}\left[\, \log p_{\mathrm{x}}(\mathbf{x}) - \log p^*_{\mathrm{x}}(\mathbf{x}) \,\right] \\ &= \mathbb{E}_{p_{\mathrm{x}}(\mathbf{x})}\left[\, \log p_{\mathrm{u}}\big(T^{-1}(\mathbf{x})\big) + \log \left|\det J_{T^{-1}}(\mathbf{x})\right| - \log p^*_{\mathrm{x}}(\mathbf{x}) \,\right] \\ &= \mathbb{E}_{p_{\mathrm{u}}(\mathbf{u})}\left[\, \log p_{\mathrm{u}}(\mathbf{u}) - \log \left|\det J_{T}(\mathbf{u})\right| - \log p^*_{\mathrm{x}}(T(\mathbf{u})) \,\right] \\ &= \mathbb{E}_{p_{\mathrm{u}}(\mathbf{u})}\left[\, \log p_{\mathrm{u}}(\mathbf{u}) - \log p^*_{\mathrm{u}}(\mathbf{u}) \,\right] \\ &= D_{\mathrm{KL}}\left[\, p_{\mathrm{u}}(\mathbf{u}) \,\|\, p^*_{\mathrm{u}}(\mathbf{u}) \,\right]. \end{aligned} \tag{125}$$

Appendix B. Constructing Linear Flows

A common approach to efficiently parameterizing invertible matrices is via *matrix decompositions*. The idea is to decompose the matrix $\mathbf{W}$ into a product of structured matrices, each of which can be easily constrained to be invertible and has a $\mathcal{O}\big(D^2\big)$ inverse and $\mathcal{O}(D)$ determinant. Below we discuss a few such parameterizations.

PLU flows Every $D \times D$ matrix $\mathbf{W}$ can be written as follows:

$$\mathbf{W} = \mathbf{PLU}, \tag{126}$$

where $\mathbf{P}$ is a permutation matrix, $\mathbf{L}$ is lower triangular, $\mathbf{U}$ is upper triangular, and all three are of size $D \times D$. We can easily constrain $\mathbf{W}$ to be invertible by restricting $\mathbf{L}$ and $\mathbf{U}$ to have positive diagonal entries. In that case, the absolute determinant of $\mathbf{W}$ can be computed in $\mathcal{O}(D)$ time by:

$$\left|\det \mathbf{W}\right| = \prod_{i=1}^{D} L_{ii} U_{ii}. \tag{127}$$

Moreover, the linear system $\mathbf{PLUz} = \mathbf{z}'$ can be solved by (a) undoing the permutation, (b) solving a lower-triangular system, and (c) solving an upper-triangular system. Solving triangular systems can be done in time $\mathcal{O}\big(D^2\big)$ by forward/backward substitution, hence inverting the PLU flow can also be done in time $\mathcal{O}\big(D^2\big)$. In practice, $\mathbf{P}$ is typically fixed to a chosen or random permutation, and $\mathbf{L}$ and $\mathbf{U}$ are learned.

Multiplication with a triangular matrix can be viewed as a special case of an autoregressive flow with affine transformer and linear conditioner. Hence, the PLU flow is simply a composition of two such autoregressive flows with opposite order, followed by a permutation. Linear flows based on the PLU decomposition have been proposed by e.g. Oliva et al. (2018); Kingma and Dhariwal (2018). A similar decomposition in the case where $\mathbf{W}$ is a convolution operator was proposed by Hoogeboom et al. (2019b) and was termed *emerging convolution*.

QR flows Another way to decompose a $D \times D$ matrix is using the QR decomposition as follows:

$$\mathbf{W} = \mathbf{QR}, \tag{128}$$

where $\mathbf{W}$ is an orthogonal matrix and $\mathbf{R}$ is upper triangular, both of size $D \times D$. In fact, we can parameterize any invertible matrix using the QR decomposition, even if we restrict the diagonal entries of $\mathbf{R}$ to be positive. In that case, the absolute determinant of $\mathbf{W}$ is given in $\mathcal{O}(D)$ time by:

$$|\det \mathbf{W}| = \prod_{i=1}^{D} R_{ii}. \tag{129}$$

Inverting the linear system $\mathbf{QRz} = \mathbf{z}'$ can be done in $\mathcal{O}(D^2)$ time by first multiplying by $\mathbf{Q}^\top$ and then solving a triangular system by forward/backward substitution. We can also think of the QR flow as a linear autoregressive flow followed by an orthogonal flow; in the next paragraph we will discuss in more detail how to parameterize an orthogonal flow. The QR flow was proposed by Hoogeboom et al. (2019b).

Orthogonal flows This is a special case of a linear flow where $\mathbf{W} = \mathbf{Q}$ is an orthogonal matrix, i.e. a matrix whose columns form an orthonormal basis in $\mathbb{R}^D$. Orthogonal flows are volume preserving since $|\det \mathbf{Q}| = 1$ and trivially invertible since $\mathbf{Q}^{-1} = \mathbf{Q}^\top$. They generalize permutations (every permutation matrix is orthogonal), and can be used on their own or in composition with other structured matrices as in the QR flow above.

Nonetheless, an effective parameterization of orthogonal matrices can be challenging (see e.g. Shepard et al., 2015; Lezcano-Casado and Martínez-Rubio, 2019, for reviews on the subject). As we discussed earlier, either the orthogonal matrices with determinant 1 or those with determinant -1 can be continuously parameterized, but not both. In the normalizing-flows literature, the following parameterizations have been explored, each with their own strengths an weaknesses.

- *The exponential map* (Golinski et al., 2019). Given a skew-symmetric matrix $\mathbf{A}$, i.e. a $D \times D$ matrix such that $\mathbf{A}^\top = -\mathbf{A}$, the matrix exponential $\mathbf{Q} = \exp \mathbf{A}$ is always an orthogonal matrix with determinant 1. Moreover, any orthogonal matrix with determinant 1 can be written this way. However, computing the matrix exponential takes in general $\mathcal{O}(D^3)$ time, so this parameterization is only suitable for small-dimensional data.
- *The Cayley map* (Golinski et al., 2019). Again given a skew-symmetric matrix $\mathbf{A}$ as above, the matrix $\mathbf{Q} = (\mathbf{I} + \mathbf{A})(\mathbf{I} - \mathbf{A})^{-1}$ is always orthogonal with determinant 1. However, this

parameterization also takes $\mathcal{O}(D^3)$ time to compute, and can only parameterize those orthogonal matrices that don't have -1 as an eigenvalue.

- *Householder transformations* (Tomczak and Welling, 2016). Any orthogonal transformation in $\mathbb{R}^D$ can be expressed as a composition of at most D reflections. A single reflection about a hyperplane perpendicular to a non-zero vector $\mathbf{v}_k$, known as a *Householder transformation*, is given by the following matrix:

$$\mathbf{H}_k = \mathbf{I} - 2\frac{\mathbf{v}_k\mathbf{v}_k^\top}{\|\mathbf{v}_k\|^2}. \tag{130}$$

  Hence, an orthogonal matrix can be parameterized by a product of K such matrices, i.e. $\mathbf{Q} = \prod_{k=1}^{K}\mathbf{H}_k$, where K doesn't need to be greater than D. Each Householder matrix has determinant -1, so the determinant of $\mathbf{Q}$ is $(-1)^K$. Each Householder transformation can be computed in time $\mathcal{O}(D)$, so an orthogonal transformation parameterized this way can be computed in time $\mathcal{O}(KD)$. The Householder parameterization is not unique and contains saddle points (for example, any permutation of the vectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_K\}$ gives the same orthogonal matrix) which can make optimization harder.

References

Justin Alsing, Benjamin D. Wandelt, and Stephen M. Feeney. Massive optimal data compression and density estimation for scalable, likelihood-free inference in cosmology. *Monthly Notices of the Royal Astronomical Society*, 477(3):2874–2885, 2018.

Lynton Ardizzone, Carsten Lüth, Jakob Kruse, Carsten Rother, and Ullrich Köthe. Guided image generation with conditional invertible neural networks. *ArXiv preprint arXiv:1907.02392*, 2019.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 214–223, 2017.

Matthias Bauer and Andriy Mnih. Resampled priors for variational autoencoders. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, pages 66–75, 2019.

Mark A. Beaumont. Approximate Bayesian computation in evolution and ecology. *Annual Review of Ecology, Evolution, and Systematics*, 41(1):379–406, 2010.

Mark A. Beaumont, Wenyang Zhang, and David J. Balding. Approximate Bayesian computation in population genetics. *Genetics*, 162:2025–2035, 2002.

Jens Behrmann, Will Grathwohl, Ricky T. Q. Chen, David K. Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. In *Proceedings of the 36th International Conference on Machine Learning*, pages 573–582, 2019.

Yoshua Bengio and Samy Bengio. Modeling high-dimensional discrete data with multi-layer neural networks. In *Advances in Neural Information Processing Systems*, pages 400–406, 2000.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *ArXiv Preprint arXiv:1308.3432*, 2013.

Mikołaj Bińkowski, Dougal J. Sutherland, Michael Arbel, and Arthur Gretton. Demystifying MMD GANs. In *International Conference on Learning Representations*, 2018.

David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

Vladimir I. Bogachev. *Measure Theory*. Springer Berlin Heidelberg, 2007.

Vladimir I. Bogachev, Alexander V. Kolesnikov, and Kirill V. Medvedev. Triangular transformations of measures. *Sbornik: Mathematics*, 196(3):309–335, 2005.

Johann Brehmer, Kyle Cranmer, Gilles Louppe, and Juan Pavez. Constraining effective field theories with machine learning. *Physical Review Letters*, 121(11):111801, 2018.

Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. The Prindle, Weber and Schmidt Series in Mathematics. PWS-Kent Publishing Company, fourth edition, 1989.

Guillaume Carlier, Alfred Galichon, and Filippo Santambrogio. From Knothe's transport to Brenier's map and a continuation method for optimal transport. *SIAM Journal on Mathematical Analysis*, 41(6):2554–2576, 2010.

Ricky T. Q. Chen and David K. Duvenaud. Neural networks with cheap differential operators. In *Advances in Neural Information Processing Systems*, 2019.

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K. Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pages 6571–6583, 2018.

Ricky T. Q. Chen, Jens Behrmann, David K. Duvenaud, and Jörn-Henrik Jacobsen. Residual flows for invertible generative modeling. In *Advances in Neural Information Processing Systems*, 2019.

Scott Saobing Chen and Ramesh A. Gopinath. Gaussianization. In *Advances in Neural Information Processing Systems*, pages 423–429, 2000.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1724–1734, 2014.

Earl A. Coddington and Norman Levinson. *Theory of ordinary differential equations*. International Series in Pure and Applied Mathematics. McGraw-Hill, 1955.

Rob Cornish, Anthony L. Caterini, George Deligiannidis, and Arnaud Doucet. Localised generative flows. *ArXiv Preprint arXiv:1909.13833*, 2019.

Kyle Cranmer, Johann Brehmer, and Gilles Louppe. The frontier of simulation-based inference. *Proceedings of the National Academy of Sciences*, 2020. doi: 10.1073/pnas.1912789117.

Ivo Danihelka, Balaji Lakshminarayanan, Benigno Uria, Daan Wierstra, and Peter Dayan. Comparison of maximum likelihood and GAN-based training of Real NVPs. *ArXiv Preprint arXiv:1705.05263*, 2017.

Nicola De Cao, Ivan Titov, and Wilker Aziz. Block neural autoregressive flow. In *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence*, 2019.

Zhiwei Deng, Megha Nawhal, Lili Meng, and Greg Mori. Continuous graph flow. *ArXiv Preprint arXiv:1908.02436*, 2019.

Peter J. Diggle and Richard J. Gratton. Monte Carlo methods of inference for implicit statistical models. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 193–227, 1984.

Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: Non-linear independent components estimation. *ICLR Workshop Track*, 2015.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using Real NVP. In *International Conference on Learning Representations*, 2017.

Laurent Dinh, Jascha Sohl-Dickstein, Razvan Pascanu, and Hugo Larochelle. A RAD approach to deep mixture models. *ICLR Workshop on Deep Generative Models for Highly Structured Data*, 2019.

Hadi M. Dolatabadi, Sarah Erfani, and Christopher Leckie. Invertible generative modeling using linear rational splines. In *Proceedings of the 23nd International Conference on Artificial Intelligence and Statistics*, 2020.

Simon Duane, Anthony D. Kennedy, Brian J. Pendleton, and Duncan Roweth. Hybrid Monte Carlo. *Physics Letters B*, 195(2):216–222, 1987.

Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural ODEs. In *Advances in Neural Information Processing Systems*, 2019.

Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. Cubic-spline flows. *ICML Workshop on Invertible Neural Networks and Normalizing Flows*, 2019a.

Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. Neural spline flows. In *Advances in Neural Information Processing Systems*, 2019b.

Gal Elidan. Copulas in machine learning. In *Copulae in Mathematical and Quantitative Finance*, pages 39–60, 2013.

Luca Falorsi, Pim de Haan, Tim R. Davidson, and Patrick Forré. Reparameterizing distributions on Lie groups. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, pages 3244–3253, 2019.

Brendan J Frey. *Graphical models for machine learning and digital communication*. MIT press, 1998.

Jerome H. Friedman. Exploratory projection pursuit. *Journal of the American Statistical Association*, 82(397):249–266, 1987.

Mevlana C. Gemici, Danilo Jimenez Rezende, and Shakir Mohamed. Normalizing flows on Riemannian manifolds. *NeurIPS Workshop on Bayesian Deep Learning*, 2016.

Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. MADE: Masked autoencoder for distribution estimation. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 881–889, 2015.

Adam Golinski, Mario Lezcano-Casado, and Tom Rainforth. Improving normalizing flows via better orthogonal parameterizations. *ICML Workshop on Invertible Neural Networks and Normalizing Flows*, 2019.

Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. The reversible residual network: Backpropagation without storing activations. In *Advances in Neural Information Processing Systems*, pages 2214–2224, 2017.

Pedro J. Gonçalves, Jan-Matthis Lueckmann, Michael Deistler, Marcel Nonnenmacher, Kaan Öcal, Giacomo Bassetto, Chaitanya Chintaluri, William F. Podlaski, Sara A. Haddad, Tim P. Vogels, David S. Greenberg, and Jakob H. Macke. Training deep neural density estimators to identify mechanistic models of neural dynamics. *Elife*, 9, 2020. doi: 10.7554/eLife.56261.

Henry Gouk, Eibe Frank, Bernhard Pfahringer, and Michael Cree. Regularisation of neural networks by enforcing Lipschitz continuity. *ArXiv Preprint arXiv:1804.04368*, 2018.

Will Grathwohl, Ricky T. Q. Chen, Jesse Betterncourt, Ilya Sutskever, and David K. Duvenaud. FFJORD: Free-form continuous dynamics for scalable reversible generative models. In *International Conference on Learning Representations*, 2019.

Alex Graves. Generating sequences with recurrent neural networks. *ArXiv Preprint arXiv:1308.0850*, 2013.

David S. Greenberg, Marcel Nonnenmacher, and Jakob H. Macke. Automatic posterior transformation for likelihood-free inference. In *Proceedings of the 36th International Conference on Machine Learning*, pages 2404–2414, 2019.

Aditya Grover, Manik Dhar, and Stefano Ermon. Flow-GAN: Combining maximum likelihood and adversarial learning in generative models. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, 2018.

Tuomas Haarnoja, Kristian Hartikainen, Pieter Abbeel, and Sergey Levine. Latent space policies for hierarchical reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, pages 1851–1860, 2018.

Junxian He, Graham Neubig, and Taylor Berg-Kirkpatrick. Unsupervised learning of syntactic structure with invertible neural projections. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1292–1302, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Gustav Eje Henter, Simon Alexanderson, and Jonas Beskow. MoGlow: Probabilistic and controllable motion synthesis using normalising flows. *ArXiv Preprint arXiv:1905.06598*, 2019.

Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving flow-based generative models with variational dequantization and architecture design. In *Proceedings of the 36th International Conference on Machine Learning*, pages 2722–2730, 2019.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Matthew Hoffman, Pavel Sountsov, Joshua V. Dillon, Ian Langmore, Dustin Tran, and Srinivas Vasudevan. NeuTra-lizing bad geometry in Hamiltonian Monte Carlo using neural transport. *ArXiv Preprint arXiv:1903.03704*, 2019.

Shion Honda, Hirotaka Akita, Katsuhiko Ishiguro, Toshiki Nakanishi, and Kenta Oono. Graph residual flow for molecular graph generation. *ArXiv Preprint arXiv:1909.13521*, 2019.

Emiel Hoogeboom, Jorn W. T. Peters, Rianne van den Berg, and Max Welling. Integer discrete flows and lossless compression. In *Advances in Neural Information Processing Systems*, 2019a.

Emiel Hoogeboom, Rianne Van Den Berg, and Max Welling. Emerging convolutions for generative normalizing flows. In *Proceedings of the 36th International Conference on Machine Learning*, pages 2771–2780, 2019b.

Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural autoregressive flows. In *Proceedings of the 35th International Conference on Machine Learning*, pages 2078–2087, 2018.

Michael F. Hutchinson. A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines. *Communications in Statistics—Simulation and Computation*, 19 (2):433–450, 1990.

Aapo Hyvärinen and Petteri Pajunen. Nonlinear independent component analysis: Existence and uniqueness results. *Neural Networks*, 12(3):429–439, 1999.

David Inouye and Pradeep Ravikumar. Deep density destructors. In *Proceedings of the 35th International Conference on Machine Learning*, pages 2167–2175, 2018.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 448–456, 2015.

Jörn-Henrik Jacobsen, Arnold W. M. Smeulders, and Edouard Oyallon. i-RevNet: Deep invertible networks. In *International Conference on Learning Representations*, 2018.

Jörn-Henrik Jacobsen, Jens Behrmann, Richard Zemel, and Matthias Bethge. Excessive invariance causes adversarial vulnerability. In *International Conference on Learning Representations*, 2019.

Priyank Jaini, Kira A. Selby, and Yaoliang Yu. Sum-of-squares polynomial flow. In *Proceedings of the 36th International Conference on Machine Learning*, pages 3009–3018, 2019.

Lifeng Jin, Finale Doshi-Velez, Timothy Miller, Lane Schwartz, and William Schuler. Unsupervised learning of PCFGs with normalizing flow. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2442–2452, 2019.

Richard M. Johnson. The minimal transformation to orthonormality. *Psychometrika*, 31: 61–66, 03 1966.

Nal Kalchbrenner, Aäron van den Oord, Karen Simonyan, Ivo Danihelka, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu. Video pixel networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1771–1779, 2017.

Sungwon Kim, Sang-Gil Lee, Jongyoon Song, Jaehyeon Kim, and Sungroh Yoon. FloWaveNet : A generative flow for raw audio. In *Proceedings of the 36th International Conference on Machine Learning*, pages 3370–3378, 2019.

Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1×1 convolutions. In *Advances in Neural Information Processing Systems*, pages 10215–10224, 2018.

Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In *International Conference on Learning Representations*, 2014a.

Diederik P. Kingma and Max Welling. Efficient gradient-based inference through transformations between Bayes nets and neural nets. In *Proceedings of the 31st International Conference on Machine Learning*, pages 1782–1790, 2014b.

Diederik P. Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *Neural Information Processing Systems*, pages 4743–4751, 2016.

Shoshichi Kobayashi and Katsumi Nomizu. *Foundations of differential geometry*, volume 1. Interscience Publishers, 1963.

Ivan Kobyzev, Simon Prince, and Marcus A. Brubaker. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020. doi: 10.1109/TPAMI.2020.2992934.

Jonas Köhler, Leon Klein, and Frank Noé. Equivariant flows: Sampling configurations for multi-body systems with symmetric energies. *ArXiv Preprint arXiv:1910.00753*, 2019.

Manoj Kumar, Mohammad Babaeizadeh, Dumitru Erhan, Chelsea Finn, Sergey Levine, Laurent Dinh, and Diederik P. Kingma. VideoFlow: A flow-based generative model for video. *ICML Workshop on Invertible Neural Networks and Normalizing Flows*, 2019.

Valero Laparra, Gustavo Camps-Valls, and Jesús Malo. Iterative Gaussianization: From ICA to random rotations. *IEEE Transactions on Neural Networks*, 22(4):537–549, 2011.

Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, pages 29–37, 2011.

Mario Lezcano-Casado and David Martínez-Rubio. Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.

Christos Louizos and Max Welling. Multiplicative normalizing flows for variational Bayesian neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 2218–2227, 2017.

Xuezhe Ma, Xiang Kong, Shanghang Zhang, and Eduard Hovy. MaCow: Masked convolutional generative flow. In *Advances in Neural Information Processing Systems*, 2019.

Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing*, 2013.

Kaushalya Madhawa, Katushiko Ishiguro, Kosuke Nakago, and Motoki Abe. GraphNVP: An invertible flow model for generating molecular graphs. *ArXiv Preprint arXiv:1905.11600*, 2019.

Murray Marshall. *Positive polynomials and sums of squares*. American Mathematical Society, 2008.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association*, 2010.

John W. Milnor and David W. Weaver. *Topology from the differentiable viewpoint*. Princeton University Press, 1997.

Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. In *International Conference on Learning Representations*, 2018.

Shakir Mohamed and Balaji Lakshminarayanan. Learning in implicit generative models. *NeurIPS Workshop on Adversarial Training*, 2016.

Thomas Müller, Brian McWilliams, Fabrice Rousselle, Markus Gross, and Jan Novák. Neural importance sampling. *ACM Transactions on Graphics*, 38(5):145, 2019.

Eric Nalisnick, Akihiro Matsukawa, Yee Whye Teh, Dilan Gorur, and Balaji Lakshminarayanan. Hybrid models with deep and invertible features. In *Proceedings of the 36th International Conference on Machine Learning*, pages 4723–4732, 2019.

Radford M. Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 54:113–162, 2010.

Frank Noé, Simon Olsson, Jonas Köhler, and Hao Wu. Boltzmann generators: Sampling equilibrium states of many-body systems with deep learning. *Science*, 365, 2019.

Junier Oliva, Avinava Dubey, Manzil Zaheer, Barnabas Poczos, Ruslan Salakhutdinov, Eric Xing, and Jeff Schneider. Transformation autoregressive networks. In *Proceedings of the 35th International Conference on Machine Learning*, pages 3898–3907, 2018.

George Papamakarios. *Neural density estimation and likelihood-free inference*. PhD thesis, University of Edinburgh, 2019. Available at https://arxiv.org/abs/1910.13233.

George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, pages 2338–2347, 2017.

George Papamakarios, David Sterratt, and Iain Murray. Sequential neural likelihood: Fast likelihood-free inference with autoregressive flows. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, pages 837–848, 2019.

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Elsevier Science, 1988. ISBN 9781558604797.

Lev Semenovich Pontryagin. *Mathematical theory of optimal processes*. Routledge, 1962.

Ryan Prenger, Rafael Valle, and Bryan Catanzaro. WaveGlow: A flow-based generative network for speech synthesis. In *Proceedings of the 2019 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3617–3621. IEEE, 2019.

Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1530–1538, 2015.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31st International Conference on Machine Learning*, pages 1278–1286, 2014.

Danilo Jimenez Rezende, Sébastien Racanière, Irina Higgins, and Peter Toth. Equivariant Hamiltonian flows. *ArXiv Preprint arXiv:1909.13739*, 2019.

Oren Rippel and Ryan Prescott Adams. High-dimensional probability estimation with deep density models. *ArXiv Preprint arXiv:1302.5125*, 2013.

Hannes Risken. Fokker–Planck equation. In *The Fokker–Planck Equation: Methods of Solution and Applications*, pages 63–95. Springer Berlin Heidelberg, 1996.

Murray Rosenblatt. Remarks on a multivariate transformation. *The Annals of Mathematical Statistics*, 23(3):470–472, 1952.

Walter Rudin. *Principles of mathematical analysis*. International Series in Pure and Applied Mathematics. McGraw-Hill, 1976.

Walter Rudin. *Real and complex analysis*. Tata McGraw-Hill Education, 2006.

Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P. Kingma. PixelCNN++: Improving the PixelCNN with discretized logistic mixture likelihood and other modifications. In *International Conference on Learning Representations*, 2017.

Yannick Schroecker, Mel Vecerik, and Jon Scholz. Generative predecessor models for sample-efficient imitation learning. In *International Conference on Learning Representations*, 2019.

Ron Shepard, Scott R. Brozell, and Gergely Gidofalvi. The representation and parametrization of orthogonal matrices. *The Journal of Physical Chemistry A*, 119(28):7924–7939, 2015.

Abe Sklar. *Fonctions de Répartition à N Dimensions et Leurs Marges*. Université Paris, 1959.

Jiaming Song, Shengjia Zhao, and Stefano Ermon. A-NICE-MC: Adversarial training for MCMC. In *Advances in Neural Information Processing Systems*, volume 30, pages 5140–5150, 2017.

Yang Song, Chenlin Meng, and Stefano Ermon. MintNet: Building invertible neural networks with masked convolutions. In *Advances in Neural Information Processing Systems*, pages 11002–11012, 2019.

Endre Süli. Lecture notes on numerical solutions of ordinary differential equations, 2010.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

Esteban G. Tabak and Cristina V. Turner. A family of nonparametric density estimation algorithms. *Communications on Pure and Applied Mathematics*, 66(2):145–164, 2013.

Esteban G. Tabak and Eric Vanden-Eijnden. Density estimation by dual ascent of the log-likelihood. *Communications in Mathematical Sciences*, 8(1):217–233, 2010.

Lucas Theis and Matthias Bethge. Generative image modeling using spatial LSTMs. In *Advances in Neural Information Processing Systems*, pages 1927–1935, 2015.

Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. In *IEEE Information Theory Workshop*, pages 1–5. IEEE, 2015.

Michalis K. Titsias. Learning model reparametrizations: Implicit variational inference by fitting MCMC distributions. *ArXiv Preprint arXiv:1708.01529*, 2017.

Jakub M. Tomczak and Max Welling. Improving variational auto-encoders using Householder flow. *NeurIPS Workshop on Bayesian Deep Learning*, 2016.

Dustin Tran, Keyon Vafa, Kumar Krishna Agrawal, Laurent Dinh, and Ben Poole. Discrete flows: Invertible generative models of discrete data. In *Advances in Neural Information Processing Systems*, 2019.

Benigno Uria, Iain Murray, and Hugo Larochelle. RNADE: The real-valued neural autoregressive density-estimator. In *Advances in Neural Information Processing Systems*, pages 2175–2183, 2013.

Benigno Uria, Iain Murray, and Hugo Larochelle. A deep and tractable density estimator. In *Proceedings of the 31st International Conference on Machine Learning*, pages 467–475, 2014.

Benigno Uria, Marc-Alexandre Côté, Karol Gregor, Iain Murray, and Hugo Larochelle. Neural autoregressive distribution estimation. *Journal of Machine Learning Research*, 17 (205):1–37, 2016.

Rianne van den Berg, Leonard Hasenclever, Jakub M. Tomczak, and Max Welling. Sylvester normalizing flows for variational inference. *The 34th Conference on Uncertainty in Artificial Intelligence*, 2018.

Rianne van den Berg, Alexey A. Gritsenko, Mostafa Dehghani, Casper Kaae Sønderby, and Tim Salimans. IDF++: Analyzing and improving integer discrete flows for lossless compression. *ArXiv Preprint arXiv:2006.12459*, 2020.

Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. WaveNet: A generative model for raw audio. *ArXiv Preprint arXiv:1609.03499*, 2016a.

Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1747–1756, 2016b.

Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with PixelCNN decoders. In *Advances in Neural Information Processing Systems*, pages 4797–4805, 2016c.

Aäron van den Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George van den Driessche, Edward Lockhart, Luis Cobo, Florian Stimberg, Norman Casagrande, Dominik Grewe, Seb Noury, Sander Dieleman, Erich Elsen, Nal Kalchbrenner, Heiga Zen, Alex Graves, Helen King, Tom Walters, Dan Belov, and Demis Hassabis. Parallel WaveNet: Fast high-fidelity speech synthesis. In *Proceedings of the 35th International Conference on Machine Learning*, pages 3918–3926, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

Cédric Villani. *Optimal transport: Old and new*, volume 338. Springer Science & Business Media, 2008.

Martin J. Wainwright and Michael I. Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1(1–2):1–305, 2008.

Prince Zizhuang Wang and William Yang Wang. Riemannian normalizing flow on variational Wasserstein autoencoder for text modeling. *ArXiv Preprint arXiv:1904.02399*, 2019.

Patrick Nadeem Ward, Ariella Smofsky, and Avishek Joey Bose. Improving exploration in soft-actor-critic with normalizing flows policies. *ICML Workshop on Invertible Neural Networks and Normalizing Flows*, 2019.

Antoine Wehenkel and Gilles Louppe. Unconstrained monotonic neural networks. In *Advances in Neural Information Processing Systems*, 2019.

Christina Winkler, Daniel E. Worrall, Emiel Hoogeboom, and Max Welling. Learning likelihoods with conditional normalizing flows. *ArXiv Preprint arXiv:1912.00042*, 2019.

Peter Wirnsberger, Andrew J. Ballard, George Papamakarios, Stuart Abercrombie, Sébastien Racanière, Alexander Pritzel, Danilo Jimenez Rezende, and Charles Blundell. Targeted free energy estimation via learned mappings. *The Journal of Chemical Physics*, 153(14):144112, 2020.

Guandao Yang, Xun Huang, Zekun Hao, Ming-Yu Liu, Serge J. Belongie, and Bharath Hariharan. PointFlow: 3D point cloud generation with continuous normalizing flows. In *Proceedings of the International Conference on Computer Vision*, 2019.

Chunting Zhou, Xuezhe Ma, Di Wang, and Graham Neubig. Density matching for bilingual word embedding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 1588–1598, 2019.

Zachary Ziegler and Alexander Rush. Latent normalizing flows for discrete sequences. In *Proceedings of the 36th International Conference on Machine Learning*, pages 7673–7682, 2019.